Efficient and Low-Cost Fault Tolerance for Web-Scale Systems

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
eines Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Dott. Marco Serafini

aus Arezzo, Italien

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Rodrigo Rodrigues, Ph.D.

Datum der Einreichung: 17. Juni 2010
Datum der mündlichen Prüfung: 16. September 2010

Darmstadt 2010
D17

Summary

Online Web-scale services are being increasingly used to handle critical personal information. The trend towards storing and managing such information on the "cloud" is extending the need for dependable services to a growing range of Web applications, from emailing, to calendars, storage of photos, or finance. This motivates the increased adoption of fault-tolerant replication algorithms in Web-scale systems, ranging from classic, strongly-consistent replication in systems such as Chubby [Bur06] and ZooKeeper [HKJR10], to highly-available weakly-consistent replication as in Amazon's Dynamo [DHJ+07] or Yahoo!'s PNUTS [CRS+08].

This thesis proposes novel algorithms to make fault-tolerant replication more efficient, available and cost effective. Although the proposed algorithms are generic, their goals are motivated by fulfilling two major needs of Web-scale systems. The first need is tolerating worst-case failures, which are also called Byzantine in the literature after the definition of [LSP82a], in order to reliably handle critical personal information. The second need is investigating proper weak consistency semantics for systems that must maximize availability and minimize performance costs and replication costs without relaxing consistency unnecessarily.

Byzantine-Fault Tolerance: There has been a recent burst of research on Byzantine-Fault Tolerance (BFT) to make it have performance and replication costs that are feasible and comparable to the fault-tolerance techniques already in use today. BFT is typically achieved through state-machine replication, which implements the abstraction of a single reliable server on top of multiple unreliable replicas [Sch90]. This line of research ultimately aimed at showing the feasibility of this approach for Web-scale systems [CKL+09] to protect these critical systems from catastrophic events such as [Das].

This thesis proposes novel algorithms to reduce the performance and replication costs of BFT. First, the thesis shows how to reduce the cost of BFT without assuming trusted components. After the seminal PBFT algorithm [CL99], a number of *fast* BFT algorithms, as for example [MA06; DGV04; KAD+07], have been proposed. These papers show the existence of an inherent tradeoff between optimal redundancy and minimal latency in presence of faulty replicas. This is problematic in Web-scale systems, where Byzantine faults are very rare but where unresponsive (benign) replicas are commonplace. This thesis proposes a novel algorithm, called Scrooge, which reduces the replication costs of fast BFT replication in presence of unresponsive replicas. Scrooge shows that the additional replication costs needed for being fast in presence of faulty replicas are only dependent on the number of tolerated Byzantine faults, and not on the number of tolerated crashes. As an implication of this result, Scrooge is optimally resilient when it is configured to tolerate one Byzantine fault and any number of crashes. Such a configuration is quite common since Byzantine faults are relatively unlikely to happen.

This thesis then explores the advantages of using trusted components. It shows that these can lead to significant latency and redundancy costs in practical asynchronous systems [SS07]. This dispelled the belief that trusted components need

to be combined with synchronous links to achieve cost reductions, as hinted by previous work [CNV04; Ver06] . This additional assumption makes previously proposed algorithms unpractical in many settings, including Web-scale systems. In three-tiered Web-scale systems, for example, one could just leverage the fact that servers in the first tier (the Web-servers) are typically more stable, standardized and less prone to vulnerabilities than application servers. The HeterTrust protocol, which is presented in this thesis, uses trusted components without assuming synchronous links. It protects data confidentiality using a number of replicas that is linear in the number of tolerated faults and has a constant time complexity. This is a significant improvement over existing approaches which do not rely on trusted component but entail quadratic redundancy costs and linear latency [YMV+03]. Furthermore, different from existing work on confidential BFT, HeterTrust uses only symmetric-key cryptography instead of public-key signatures. HeterTrust features some interesting ideas related to speculation [KAD+07] and tolerance to denial-of-service attacks [ACKL08; CWA+09] that have been further developed by work published immediately after [SS07]. In parallel to this thesis' work, the use of trusted components in asynchronous systems was also independently explored in [CMSK07].

Weak consistency: Some replicated Web-scale applications cannot afford strong consistency guarantees such as Linearizability [HW90]. The reason is the impossibility of implementing shared objects, as for example databases, that are available in presence of partitions or asynchrony [GL02]. With few exceptions, however, all these systems relax Linearizability even in periods when there are no partitions nor asynchrony and no relaxation is needed to keep the system available. Since this relaxation is problematic for many applications, recent research is focusing on stronger consistency guarantees which can be combined with high availability.

This thesis introduces a novel consistency property, called Eventual Linearizability, which allows Linearizability to be violated only for finite windows of time. This thesis also describes Aurora, an algorithm ensuring Linearizability in periods when a single leader is present in the system. Aurora is *gracefully degrading* because it uses a single failure detector and obtains different properties based on the actual strength of this failure detector, which is not known a priori. For Eventual Linearizability, a $\Diamond\mathcal{S}$ failure detector is needed. In periods of asynchrony when links are untimely and no single leader is present, Aurora gracefully degrades to Eventual Consistency [FGL+96; Vog09] and Causal Consistency [Lam78]. For these property, Aurora only relies on a strongly complete failure detector $\mathcal{C}$. In order to complete *strong* operations, which must be always linearized, a $\Diamond\mathcal{P}$ failure detector is used. This is stronger than $\Diamond\mathcal{S}$, the weakest failure detector needed to implement consensus [CHT96], and thus linearizable shared objects. This thesis shows that there exists an inherent cost in combining Eventual Linearizability with Linearizability.

Kurzfassung

Web-basierte Online-Dienste beinhalten in zunehmendem Maße die Verarbeitung sensibler personenbezogener Daten. Die steigende Tendenz, solche Daten in der “Cloud” zu speichern und zu verwalten, erhöht den Bedarf verlässlicher Realisierungen dieser Funktionen für eine steigende Anzahl Web-basierter Anwendungen, wie etwa E-Mail, Kalender, Fotoalben oder Online-Banking. Dieser Trend erklärt die zunehmende Verwendung fehlertoleranter Replikationsalgorithmen bei der Implementierung Web-basierter Anwendungen. Die zur Anwendung kommenden Implementierungen reichen von klassischer, stark konsistenter Replikation in Systemen wie Chubby [Bur06] und ZooKeeper [HKJR10] hin zu hochverfügbarer, schwach konsistenter Replikation, etwa in Amazons Dynamo [DHJ$^+$07] oder Yahoo!s PNUTS [CRS$^+$08].

Die vorliegende Arbeit stellt neuartige Algorithmen für fehlertolerante Replikation vor, mit dem Ziel die Effizienz, Verfügbarkeit und Wirtschaftlichkeit dieser Mechanismen zu erhöhen. Wenngleich die vorgestellten Algorithmen allgemein anwendbar sind, erfüllen sie zwei Eigenschaften, die wesentlich durch den Einsatz in Web-basierten Systemen motiviert sind. Die erste Eigenschaft ist die Toleranz von Worstcase-Fehlern, in der Literatur auch als “Byzantine” [LSP82a] bezeichnet, um eine zuverlässige Verarbeitung sensibler personenbezogener Daten zu gewährleisten. Die zweite Eigenschaft ist die Entwicklung einer geeigneten Semantik schwacher Konsistenz für Systeme, für die höchstmögliche Verfügbarkeit und geringstmöglicher Zusatzaufwand hinsichtlich Performanz und Replikation sicherzustellen, Abschwächungen der Konsistenz aber weitgehend zu vermeiden sind.

Toleranz von “Byzantine” Fehlern: Die Toleranz von “Byzantine” Fehlern (englisch Byzantine Fault Tolerance, BFT) wurde kürzlich zum Gegenstand intensivierter Forschung mit dem vordergründigen Ziel, ihren implizierten Zusatzaufwand (bzgl. Performanz und erforderlicher Replikation) auf ein Maß zu reduzieren, das mit dem herkömmlicher Fehlertoleranzmechanismen vergleichbar ist. BFT wird zumeist durch die Replikation von Zustandsautomaten erzielt, indem die Illusion eines einzelnen zuverlässigen Servers durch die (für den Nutzer transparente) Koordination mehrerer unzuverlässiger Server erzeugt wird [Sch90]. Als ultimatives Ziel dieser Forschungsrichtung ist die Anwendbarkeit dieses Ansatzes für Web-basierte Systeme zu sehen [CKL$^+$09], um die so implementierten kritischen Anwendungen vor folgenschwerem Fehlverhalten, wie es etwa in [Das] beschrieben ist, zu schützen.

Die vorliegende Arbeit stellt neue Algorithmen vor, die den Performanz- und Replikationsaufwand von BFT reduzieren. Zunächst wird gezeigt, wie dieses Ziel ohne die Annahme vertrauenswürdiger Komponenten erreicht werden kann. Nach der Vorstellung des einflussreichen PBFT-Algorithmus [CL99] wurde eine Reihe *schneller* BFT-Algorithmen, wie zum Beispiel [MA06; DGV04; KAD$^+$07] entwickelt. Diese Arbeiten zeigen unter der Annahme fehlerbehafteter Repliken einen inhärenten Kompromiss zwischen optimaler Redundanz und minimaler Latenz auf.

In Web-basierten Systemen, in denen "Byzantine" Fehler nur selten, Ausfälle von Repliken hingegen häufig auftreten, stellt sich dieser unvermeidbare Kompromiss als problematisch heraus. Der in dieser Arbeit vorgestellte Algorithmus "Scrooge" reduziert den Replikationsaufwand schneller BFT-Replikation in Gegenwart nicht reagierender Repliken. Scrooge zeigt, dass der zusätzliche Replikationsaufwand zur Erzielung einer höheren Geschwindigkeit ausschließlich von der Anzahl der zu tolerierenden fehlerbehafteten Repliken abhängt und nicht von der Anzahl zu tolerierender Ausfälle. Als Konsequenz erzielt Scrooge optimale Robustheit für die Toleranz eines einzelnen "Byzantine"-Fehlers und einer beliebigen Anzahl von Ausfällen. Solche Szenarien sind charakteristisch für Web-basierte Systeme, in denen "Byzantine"-Fehler selten sind.

Anschließend daran untersucht die vorliegende Arbeit potenzielle Vorteile der Verwendung vertrauenswürdiger Komponenten. Es wird gezeigt, dass diese zu einer signifikanten Reduktion der Latenz und durch Redundanz verursachten Kosten in anwendungstypischen asynchronen Systemen führen können [SS07]. Dies verwirft die These früherer Arbeiten [CNV04; Ver06], dass eine Kostenreduktion durch vertrauenswürdige Komponenten zwingend die Verfügbarkeit synchroner Kommunikationskanäle erfordert. Diese zusätzliche Forderung nach Synchronität führt zu einer deutlichen Beschränkung möglicher Einsatzgebiete bestehender Lösungen, beispielsweise in Web-basierten Systemen. In dreistufig organisierten Web-basierten Systemen, zum Beispiel, kann man sich zunutze machen, dass Server in der ersten Ebene des Systems (die Webserver) üblicherweise standardisiert, stabiler und weniger fehleranfällig sind als beispielsweise Application-Server. Der "HeterTrust" Protokoll, der in dieser These eingeführt wird, erfordert eine zur Anzahl der zu tolerierenden Fehler lineare Anzahl von Repliken um die Vertraulichkeit von Daten sicher zu stellen, und hat konstante Komplexität. Dies ist eine Deutliche Verbesserung gegenüber bestehenden Ansätzen, die zwar keine vertrauenswürdigen Komponenten erfordern, aber quadratische Redundanzkosten und lineare Latenzen mit sich bringen [YMV+03]. Ebenfalls im Gegensatz zu anderen die Vertraulichkeit berücksichtigenden BFT-Ansätzen verwendet HeterTrust symmetrische Kryptoverfahren anstelle von Public-Key-Verfahren. HeterTrust beinhaltet einige interessante Ideen in den Bereichen der Spekulation [KAD+07] und der Toleranz von Denial-of-Service-Angriffen [ACKL08; CWA+09], deren Eigenschaften in weiteren Arbeiten untersucht und in unmittelbarer Folge von [SS07] publiziert wurden. In der selben Zeit wie der vorliegende Arbeit wurde die Verwendung vertrauenswürdiger Komponenten in asynchronen Systemen unabhängig in [CMSK07] untersucht.

Schwache Konsistenz: Für einige Web-basierte Anwendungen ist die Zusicherung starker Konsistenzeigenschaften wie Linearisierbarkeit nicht möglich [HW90]. Die Ursache dafür liegt in der Unmöglichkeit einer Implementierung von "Shared Objects", wie zum Beispiel Databases, in Fällen von Partitionierung oder Asynchronität [GL02]. Allerdings geben bis auf wenige Ausnahmen alle diese Systeme Linearisierbarkeit auch in Betriebsabschnitten auf, in denen weder Par-

titionierung, noch Asynchronität vorliegen. Da dieser Lockerung der Konsistenz für einige Anwendungen problematisch ist, konzentriert sich neuliche Forschung auf stärkere Konsistenzeigenschaften, die sich mit Hochverfügbarkeit kombinieren lassen.

Die vorliegende Arbeit führt "Eventual Linearizability" als neue Konsistenzeigenschaft ein, die eine Verletzung der Linearisierbarkeit für endliche Zeitabschnitte gestattet. Sie beschreibt weiterhin Aurora, einen Algorithmus zur Sicherstellung von Linearisierbarkeit in Phasen, in denen ein einzelner Leader im System vorhanden ist. Die Leistungsfähigkeit von Aurora vermindert sich schrittweise im Falle sich verschlechternder Ausführungsbedingungen. Aurora verwendet einen einzelnen a priori nicht näher bestimmten Fehlerdetektor, von dessen Stärke aber Eigenschaften Auroras abhängen. "Eventual Linearizability" erfordert einen $\Diamond\mathcal{S}$ Fehlerdetektor. In Phasen von Asynchronität, in denen die Pünktlichkeit von Nachrichten und die Präsenz eines einzelnen Leaders nicht gewährleistet werden kann, reduziert sich die von Aurora getroffene Zusicherung auf "Eventual Consistency" [FGL$^+$96; Vog09] und "Causal Consistency" [Lam78]. Für diese Eigenschaften benötigt Aurora lediglich einen Fehlerdetektor $\mathcal{C}$ mit "Strongly Complete"-Eigenschaft. Für die Durchführung sogenannter "Strong Operations", die "Linearizability" erfordern, wird ein $\Diamond\mathcal{P}$ Fehlerdetektor verwendet. Dieser ist stärker als $\Diamond\mathcal{S}$, welches der schwächste Fehlerdetektor für die Implementierung von "Consensus" ist [CHT96] und somit auch "Linearizable Shared Objects". Die vorliegende Arbeit zeigt, dass ein inhärenter Aufwand bei der Kombination von "Eventual Linearizability" und "Linearizability" existiert.

Acknowledgements

When I was I kid and people asked me what I would have liked to do once grown-up, I always said that I wanted to become like Gyro Gearloose and invent marvelous machines. But I was not really serious, and for most of my life I just fancied about becoming a researcher, among many other things. There have been twists and turns on the way to get here.

I might owe my choice of becoming a computer scientist to my friend Lorenzo. We were children, and during an endless summer on the Tuscan countryside he showed me his new toy: a Commodore 64. It was the fist machine I saw that you could have actually hacked! But all he did with it was inserting videogame tapes and pressing play to load them. I promised myself that I one day I would have learned how computers really work.

My parents have kept a loving eye on me, supporting me without ever being oppressive. They had imagined a different future for me, working on their side, but they always gave me a chance to do things my way, even when it was not clear what I was up to. Now they are proud of my choices and that is the best reward ever. Thanks a lot!

A big twist was talking to Neeraj in Florence, on a June afternoon. By inviting me to join his group, he introduced me to a profession that still seems too good to be true. He made me a great gift: the total freedom to pursue whatever topic I found exciting, learning from my own failures. I had to struggle, but it has paid off.

Many friends and colleagues made my life in Darmstadt easier and contributed to my personal and technical growth. It is fun to work and to be friend with Péter, our trips to the Zoo were indeed very cool. I had no doubt when I chose him as best man for my wedding. Dan is a great friend who helped me a lot to get acquainted to Germany. By stopping by, talking about his ideas, and being critical towards mine, he was fundamental in letting me rediscover my early love for theoretical computer science. Andreas, Piotr, Matthias, Dinu, Vinay, Birgit, Sabine, and all the other DEEDS folks made the working place a special, fun place.

I was lucky enough to get feedback from great senior researchers such as Cristian Cachin, Rachid Guerraoui, Flavio Junqueira, Stefan Katzenbeisser, András Pataricza, Rodrigo Rodrigues, Fred Schneider, Helmut Veith. I appreciated the value of the time they dedicated to my work.

The best result of my PhD was definitively meeting Ilaria. That, alone, would have made graduating worth it.

Marco Serafini
Barcelona, June 17, 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Online applications and services are becoming ubiquitous. Their convenience, easy accessibility and potential for integration are convincing many users to increasingly entrust these services with critical data, as for example emails, pictures, backup files, financial information, and many others. Because of the immense potential audience such services have, they need to face scalability issues that were never faced by previous computer systems. Major Internet companies run their infrastructure on datacenters with tens of thousands of processors connected by sophisticated high-speed network infrastructures. In these systems, which we call Web-scale, even failures that would be quite unlikely in a system with few nodes become frequent and unavoidable. This explains the popularity of techniques for tolerating faults in these systems.

Fault tolerance in distributed systems is typically achieved through *replication*. The same logical functionality is replicated over multiple physical nodes, which are kept consistent using a replication algorithm.

Web-scale systems pose significant challenges for replication. Replicated services must support a high volume of requests with low latency. For example, a ZooKeeper cluster is typically shared by multiple large-scale applications, and it is critical that it does not become a performance bottleneck [HKJR10]. Furthermore, since multiple instances of a replicated service are deployed for scalability, it is important to reduce the number of replicas of each instance. Crashes in these systems are also very common, and should not result in a degradation of performance [DHJ+07]. Finally, these applications have strict latency and availability requirements, and they sometime decide to trade consistency for these goals [GGL03; DHJ+07; CRS+08].

Efficient Byzantine-Fault Tolerance The increased adoption of replication techniques in Web-scale systems has led to an interesting contamination between theory and practice of distributed computing. One direction where this contamination has been particularly promising is Byzantine-Fault Tolerance (BFT). The goal of BFT is to increase the reliability of replicated applications by making them tolerant to worst-case faults that cannot be easily detected using error detection. These faults are becoming more and more relevant. It is known, for example, that modern hardware tends to have lower reliability due to decreased feature size [Con02; Bor05]. Storage components in large scale systems also have quite high failure rates [PWB07; SG07]. As a result, there are relatively many cases of Web-scale systems becoming unavailable due to undetected errors, the most notable cases being the July 2008 outage of the Amazon S3 service [Das] or the July 2009 outage of the Google File System [SJR09].

BFT uses the so-called state-machine approach [Sch90] to mask replicas returning arbitrary results through voting. State-machine replication implements the abstraction of a logical reliable server using a set of unreliable replicas. A consensus algorithm is used by replicas to agree a consistent order of execution for all requests. This enables voting on the results of the requests by ensuring that all correct replicas output the same result for each request. Simple and practical techniques for state-machine replication to tolerate crashes, such as the Paxos algorithm, are well known [Lam98]. Despite early theoretical results [BT85], tolerance to Byzantine faults was thought to be practically unfeasible in asynchronous systems until Castro and Liskov proposed their PBFT protocol [CL99]. PBFT resembles Paxos and does not use public-key cryptography. A number of subsequent paper showed how to make BFT more efficient and practical (for example [CML+06; AEMGG+05; DGV04; MA06; KAD+07]), including recent work on applying BFT to popular Web-scale systems [CKL+09]. Another line of work has explored how to reduce the replication costs of BFT using trusted components with a restricted failure mode (for example [CVN02; CNV04; Ver06; CMSK07; LDLM09]).

Weak consistency Another direction where fault-tolerance in Web-scale systems has potential for improvement is the use of weak consistency semantics. State-machine replication implements one of the strongest known form of consistency, Linearizability [HW90]. As discussed, this entails solving consensus, a problem whose intrinsic complexity is quite high because it requires the availability of a single leader process that can communicate with a majority of correct processes [CHT96; CT96]. The latency requirements of some Web-scale applications, however, are so strict that using consensus would result in frequent timing failures, i.e., in the system being unavailable.

The latency problem is further exacerbated by the presence of partitions.

Unavailability in widea-area-network (WAN) links is generally recognized as an important issue (see for example [DCGN03] for an experimental evaluation of the problem). However, it has been reported that partitions occur even within datacenters, most likely due to failures of network devices [Vog09]. Applications that need to remain available in presence of partitions trade consistency for high availability. Example of such applications are Amazon's Dynamo [DHJ+07] or Yahoo!'s PNUTS [CRS+08]. A typical a form of weak consistency, which is for example implemented by Dynamo, is Eventual Consistency [SS05; Vog09]

1.1 Open issues

This thesis tackles three open issues in building efficient fault-tolerant replication algorithms for Web-scale systems.

1.1.1 What is the Minimal Replication Cost for High-Performance BFT?

The latency, and in some cases throughput, overhead of Byzantine fault tolerant consensus can be reduced by using fast agreement algorithms, such as DGV [DGV04], FaB [MA06] or Zyzzyva [KAD+07]. The minimum required number of replicas for BFT is $2f + b + 1$, where b is the number of Byzantine failures and f the total number of failures tolerated, including also crashes [BT85; Lam03]. Current approaches can achieve the theoretically minimal latency in presence of f unresponsive replicas only if $f + b - 2$ or more additional replicas are used [MA06; DGV04; KAD+07]. Theoretical lower bound results show that these upper bounds are tight. This implies that fast agreement in presence of faulty replicas is only possible if the number of replicas used in the system is at least two times higher than the number needed for crash tolerance [DGV04; MA06]. This represents an obstacle for the adoption of BFT in Web-scale systems that need be fast even in presence of crashes or partitions, such as [GGL03; DHJ+07], because crashes can be quite frequent in large-scale systems. It must be noted that a large number of instances of a given replicated service can be deployed in a Web-scale system, and each of these instances corresponds to a BFT replication cluster. This implies that additional replication costs in each cluster can result in a significant increase of the overall costs of adopting BFT.

Another limitation of fast agreement protocols is that they can only remain fast in runs where a specific replica, called primary, is correct. If this becomes unresponsive, these algorithms suffer a performance degradation.

In other word, these algorithms are fast in *all* runs *except* those where the primary is faulty.

1.1.2 Is Using Trusted Components in BFT Systems Useful in Practice?

Fault tolerant distributed protocols typically utilize a homogeneous fault model where all processes are assumed to fail in the same manner. However, different processes can have different degrees of reliability, depending for example on their complexity or on the range of functionalities they offer to external entities. This implies that some processes may be more trustworthy than others. These trusted processes can be assumed to fail only by crashing, while other components may fail in a Byzantine manner.

Consider for example three-tiered Web-scale systems, where the first tier handling client requests consists of Web server, the second of application servers and the third of databases. It has been observed that Web servers in the first tier are typically more reliable than application servers in the second tier [ZBWM08]. The reason is that Web servers are typically generic off-the-shelf services which withstand rigorous testing. Application servers, on the contrary, often run software that cannot undergo a very through testing due to its shorter time-to-market. Another example of relatively more trustworthy nodes are network routers, which are usually more reliable than end users' PCs because they execute a restricted and well-known functionality. Unexperienced users do not typically install new and potentially malicious software on routers. Finally, trustworthy components can also be implemented as protected hardware components [Gro].

At the time when this thesis was written, some papers examined how to use trusted computing elements in the context of Byzantine-fault tolerant replication [CNV04; Ver06]. These algorithms, however, required a specific architecture with a trusted coprocessor and, more importantly, assumed the availability of a synchronous, reliable and trusted network between these trusted entities. This last assumption is hard to require even in local area networks. Furthermore, previous work only focused on integrity and did not consider the use of trusted entities to improve the confidentiality of the system. Protecting confidentiality in systems where processes can only fail in a Byzantine manner entail a latency overhead, measured in terms of communication steps, that is linear in the number of tolerated faults, and a redundancy cost that is quadratic [YMV$^+$03]. The way replication can be integrated with the protection of confidentiality in systems using trusted components was still an open issue.

1.1.3 Are There Viable Alternatives to Eventual Consistency?

When designing such replication algorithms, there is a fundamental trade-off between consistency of service state and availability. The CAP Theorem captures this trade-off (Consistency, Availability, and Partition-Tolerance: pick two [GL02]). Strong consistency guarantees simplify the task of developing applications for such systems, but have stronger requirements on the connectivity of replicas for progress. Weakly consistent replication provides higher availability, but is harder to deal with.

A strong consistency guarantee often used as a correctness property is *Linearizability* [HW90]. Linearizability ensures that all clients observe changes of the service state according to the real-time precedence order of operations and that operations are serializable. At a high level, it provides clients with the view of a single, robust server. The simplicity of this abstraction explains its popularity in Web-scale replication libraries [Bur06; HKJR10]. However, the high latency and low availability entailed in providing Linearizability motivates the use of weaker consistency semantics.

Weakly consistent replication can terminate in worst-case runs and typically has lower latency. An established weak semantic is *Eventual Consistency*: if no new operation is invoked, all replicas converge to the same state [SS05; Vog09]. Whenever concurrent operations are present, however, replicas can transition to an inconsistent state and thus violate Linearizability. This is common to weakly consistent replication algorithms [SS05]. It has been observed, however, that Eventual Consistency has several drawbacks [GHOS96; BCvR09]. This calls for more fundamental research on this topic and for a better understanding of the fundamental tradeoffs involved in building weakly consistent systems.

Eventual Consistency is too weak to solve some distributed problem. Consider for example the problem of implementing a replicated and highly-available master in a master-worker scheme. These schemes are very common in Web-scale systems, which use often a master process to partition large workloads over a large number of worker processes. If each master replica assigns task in isolation, multiple workers can execute duplicated work. Since master replicas coordinate only asynchronously, there is no limit to the amount of duplicated work done by the workers even in runs where the system is synchronous. This makes the use of Eventual Consistency for master-worker schemes unpractical. On the other hand, using Linearizability might not result in an adequate level of availability.

	Replication costs (min. $2f + b + 1$ [Lam03])	Fast w. no unresponsive replica	Fast w. f unresponsive replicas
PBFT [CL99]	$3f + 1$	NO	NO
Zyzzyva [KAD$^+$07]	$3f + 1$	YES	NO
Zyzzyva5 [KAD$^+$07]	$5f + 1$	YES	YES
DGV [DGV04]	$3f + 2b - 1$	YES	YES
Scrooge	$2f + 2b$	YES	YES

Table 1.1: Comparison of primary-based BFT replication protocols that tolerate f failures, including b Byzantine ones.

1.2 Thesis Contributions

This thesis proposes three novel algorithms addressing each of the research problems that have been identified previously.

1.2.1 Fast BFT with Unresponsive Replicas

This thesis aims at improving on the tradeoff between high performance (in terms of both throughput and latency) and redundancy costs. It proposes Scrooge, a new BFT replication algorithm that reduces the replication costs of fast BFT. Scrooge turns around existing lower bounds by providing slightly relaxed performance properties. Existing fast algorithms are fast in all runs except those where the primary is faulty, whereas Scrooge allows some performance degradation also if other replicas become faulty. This relaxation is acceptable in many systems. Another insight used in Scrooge is that the Message Authentication Codes (MACs) used in all practical BFT algorithms to authenticate messages can also be used to identify faulty replicas. Considering MACs explicitly results in a stronger system model than just assuming the use of authenticated channels, as done by other algorithms, although this is a non-assumption in practice.

The Scrooge protocol is fast in presence of f unresponsive replicas using only $b - 1$ additional replicas. It thus shows that the additional replication costs can be independent of f and thus of the number of tolerated crashes. This makes Scrooge cheaper than the cost lower bound of [DGV04] and particularly cost-effective for systems that must tolerate many crashes (large f) and few Byzantine faults (small b). When tolerance to f faults including only one Byzantine fault is sought, Scrooge achieves the minimal replication cost of $2f + 2$ and requires only one replica more than protocol tolerating f crashes only. These requirements are common in systems where Byzantine failures are only an unlikely corner case.

A comparison between Scrooge and other state machine replication protocols tolerating Byzantine faults is illustrated in Table 1.1. The first three protocols in the Table assume $f = b$. A protocol is denoted as *fast* if it has minimal best case latency to solve consensus [MA06; DGV04]. If the primary is faulty or the clients are Byzantine none of these protocols is fast. Upon failures of other replicas, Scrooge is fast after a bounded time whereas Zyzzyva5 is always fast. For DGV, the cost for $f > 1$ in order to be fast with f unresponsive replicas is depicted. For $f = 1$ the corresponding cost is $2f + 2b + 1$ replicas.

Although Scrooge can detect and isolate Byzantine failures of non-primary replicas, this thesis uses the notion of unresponsive replicas to stress that the goal of Scrooge and of the other cited algorithms is not to be fast in presence of attacks. Achieving acceptable performance in presence of worst-case attacks requires using different techniques, such as using specific network topologies, which are mostly orthogonal to the work of this thesis (see for example [ACKL08; CWA$^+$09]). However, Scrooge explicitly considers the use of public-key signatures[1] for client requests, as indicated in [CWA$^+$09], and leverages it for correctness.

The thesis includes an experimental evaluation of Scrooge. Scrooge performs as well as state-of-the-art fast BFT protocols like Zyzzyva and Zyzzyva5 if all replicas are responsive. In scenarios with at least one unresponsive replica the thesis shows that:

- The peak throughput advantage of Scrooge is more than 1.3 over Zyzzyva. Scrooge also has lower latency with high load;
- Scrooge reduces latency with low load by at least 20% and up to 98% compared to Zyzzyva;
- Scrooge performs as well as Zyzzyva5, which uses $f + 1$ more replicas than Scrooge (with $f = b$);
- As the number of tolerated faults increases, the overhead of Scrooge degrades more slowly than in other protocols using equal or lower redundancy.

1.2.2 Trusted Processors with Asynchrony

Assuming a synchronous trusted network can introduce a vulnerability in the system and make its implementation challenging. This thesis examines

[1]Note that public-key signatures are stronger and more expensive to produce and verify than MACs, since the latter are based on symmetric-key cryptography.

Protocol	**SM**	**FM**	n	**Latency**	**Confid.**	**Crypt.**
Paxos [Lam98; Lam01]	PS	C	$2g+1$	4/5	-	-
BFT [CL99]	PS	B	$3f+1$	4	no	MAC
FaB [MA06]	PS	B	$5f+1$	3	no	MAC
Correia *et.al* [CNV04]	W	W	$2m+1$	5	no	MAC
Marchetti *et.al* [MBTPV06]	PS	C	$3g+2$	4/5	-	-
Yin *et.al* [YMV+03]	PS	B	f^2+6f+2	$2f+7$	yes	TS
HeterTrust	PS	H	$2f+2g+2$	4	yes	MAC

n = lower bound on # nodes; $g/f/m$ = upper bound on # fail-crash / Byzantine / mixed nodes
SM = System Model (**P**artially **S**ynchronous / **W**ormhole)
FM = Fault Model (**C**rash / **B**yzantine / **W**ormhole / **H**eterogeneous)
MAC = Message Authentication Codes; TS = Threshold Signatures

Table 1.2: Comparison between HeterTrust and other deterministic state machine replication protocols

how processes with restricted failure mode can be leveraged to simplify BFT and to reduce the costs of obtaining confidentiality even in asynchronous networks where timeliness is only required for progress.

In order to leverage the heterogeneity of fault models in practical distributed systems, as for example in Web-scale systems, this thesis proposes a novel fail-heterogeneous architectural model for distributed systems which considers two classes of nodes: (a) full-fledged execution nodes, which can be fail-Byzantine, and (b) lightweight, validated coordination nodes, which can only be fail-crash. To illustrate how the model can be used to reduce the costs BFT, the thesis introduces HeterTrust, a practical state machine replication protocol that prevents intruded servers from disclosing confidential data. The challenge in using trusted nodes is to define the interface and the functionalities that they should offer to the rest of the system. These must be simple (to ease bug-free design and error detection), generic (to ease adoption and eliminate the need for potentially faulty extensions) and require little storage and computational capability (to make it cost effective). In fact, many coordination tasks that are typical of state machine replication have these three characteristics and can thus be abstracted and encapsulated into these trusted nodes, which are called coordinators.

In HeterTrust, coordinators are physically interposed between clients and execution servers. One of the coordinators, called the leader, has the role of proposing the order of execution of the operations to the execution servers. The execution servers execute the operations in the proposed order and send replies back to the coordinators. These check that the replies are correct by waiting for a set of consistent reply messages such that at least one correct replica has sent one of the messages. Only then is the reply forwarded to the clients. Executing this check ensures that no spurious reply, which is generated by malicious servers to leak confidential data to the clients, is ever sent to any client. Coordinators also agree on the order of the operations to handle failures of the current leader.

Compared to systems using homogeneous fault models and achieving similar goals, HeterTrust has lower latency, requires fewer execution nodes, and does not require the use of expensive asymmetric cryptography. Compared to other existing algorithms using trusted components, HeterTrust does not require synchrony in a partition of the system as required by the Wormhole model [CNV04; Ver06]. A comparison of HeterTrust with other state machine replication algorithm is in Table 1.2

Overall, the thesis presents the following contributions:

- It introduces and motivates the fail-heterogeneous architectural model, taking the problem of practical trustworthy state machine replication as a case study and presenting the HeterTrust protocol;
- It demonstrates that, by using a majority of coordination nodes with the same correct design and without a trusted synchronous network, the minimal number of replicas with diversified design to tolerate f malicious faults can be reduced from $3f + 1$ [BT85] to $2f + 1$;
- It indicates how attackers can be prevented from disclosing confidential data of intruded servers by means of simple symmetric-key cryptography and using only $2g + 1$ coordinators out of which g can fail by crashing;
- It shows that the latency overhead for replication and confidentiality with respect to a non replicated service is two communication steps;
- It discusses for the first time how to handle Denial-of-Service (DoS) attacks in BFT systems.

A relevant additional result related to the second and third contributions is the following: If trusted coordinators model subcomponents of execution servers, then the same reduction of redundancy cost achieved in [CNV04] using dedicated synchronous channels can be obtained in asynchronous systems.

Results similar to some of these contributions have been also proposed by independent research carried out in parallel with the work of this thesis. Algorithms using trusted components in asynchronous systems to reduce the cost of BFT replication were proposed in [CMSK07]. Follow up work have further explored this model, for example [LDLM09].

HeterTrust achieves latency reduction by letting servers execute requests before the order proposed by the leader is agreed upon, a technique closely related to speculation [KAD+07]. Similar to speculation, clients determine that agreement has been reached before coordinator and servers know it. This occurs when replies are delivered.[2]

Another innovative line of work introduced by this thesis and further later developed in [ACKL08; CWA+09] was the tolerance of BFT replication algorithms to DoS attacks.

1.2.3 Eventual Linearizability and Aurora

Current consistency semantics ensure Linearizability either *always* or *never*. This thesis aims at finding better tradeoffs between availability and consistency. In fact, current weakly consistent systems remain inconsistent also in periods where the system is perfectly timely and there are no partitions, which in some sense contradicts the sense of the CAP Theorem.

Weak consistency leverages the fact that in many real world applications requiring high availability, processes often issue operations that do not need to be linearized. We call these operations *weak* as opposed to *strong* operations that must be linearized. Ideally, weak operations applied to a shared object should terminate irrespective of the failure detector output or of the number of faulty processes. In a gracefully-degrading approach to weak consistency, it is acceptable that weak operations violate Linearizability when the system deviates from its "normal" behavior, but only if such violations cease when the anomalies terminate [HW87; AAL+08]. We call this property *Eventual Linearizability.*

Shared objects with Eventual Linearizability can be used, for example, in master-worker applications to replicate the master. Consider again the example of a replicated real-time queue used to dispatch taxi requests to taxi cabs [HW87]. Some degree of redundant work, such as having multiple cabs respond to the same call, can be accepted if this prevents the system from becoming unavailable in presence of anomalies, guaranteeing that cabs can always dequeue requests. However, no redundant work should take place when there is no anomaly.

A surprising result of this thesis is the answer to the following question: Is it possible to achieve these desirable properties of weak operations without sacrificing Linearizability and termination of strong operations? The thesis answers this question in the negative. In fact, combining Linearizability and Eventual Linearizability requires using a stronger failure detector to complete strong operations than the one sufficient for Consensus.

[2]Using the Paxos terminology, which is also explained in this thesis in section 2, clients act as *learners.*

This thesis introduces the notion of Eventual Linearizability for weak operations, which is the strongest known consistency property that can be attained with $\Diamond\mathcal{S}$ despite *any number* of crashes. Eventual Linearizability guarantees that Linearizability is violated only for a finite time window. It satisfies the same locality and nonblocking properties as Linearizability. An additional result is that Eventual Linearizability for weak operations cannot be provided using existing notions of Eventual Consistency [SS05; Vog09; FGL$^+$96]. With Eventual Consistency, in fact, Linearizability can be violated whenever multiple operations are invoked concurrently. Therefore, Eventual Consistency never ensures Linearizability.

This thesis also introduces a primitive, called *Eventual Consensus*, that it proves to be necessary and sufficient to implement Eventual Linearizability. Eventual Consensus is strictly weaker than Consensus, since it can be implemented with $\Diamond\mathcal{S}$ despite any number of faulty processes. Inputs to Eventual Consensus are operations proposed by processes, and outputs are sequences of operations. Informally, Eventual Consensus requires that after some unknown time t, all operations proposed after t are totally ordered at each process *before* being completed.

Beyond introducing and formalizing Eventual Linearizability and Eventual Consensus, the thesis studies whether Consensus implementations can be extended to provide Eventual Consensus without degrading their properties. It presents a shared object implementation, called Aurora, which provides Linearizability for strong operations and Eventual Linearizability for weak operations using the Eventual Consensus primitive. Aurora is gracefully degrading because it achieves different consistency properties based on the actual strength of the failure detector it uses. In particular, it degrades Eventual Linearizability to Eventual Consistency only in periods where Consensus would block due to the absence of a single leader process.

For high availability, Aurora ensures termination of weak operations in asynchronous runs. In these runs, Aurora guarantees Eventual Consistency and also causal consistency [Lam78] of weak operations. Unlike other weakly consistent implementations such as Lazy Replication [LLSG92] and Bayou [TTP$^+$95], Aurora additionally implements Eventual Linearizability for weak operations in runs where processes have access to a failure detector of class $\Diamond\mathcal{S}$. In this case, strong operations terminate in the absence of concurrent weak operations if a majority of correct processes exists. Finally, if the processes have access to a failure detector of class $\Diamond\mathcal{P}$, then all operations terminate even in presence of concurrency.

It may seem unnecessary that Aurora requires a stronger failure detector than a Consensus algorithm to terminate strong operations. This thesis shows, perhaps unexpectedly, that this reflects a fundamental tradeoff.

Specifically, the thesis shows that with $\Diamond\mathcal{S}$, it is impossible to ensure termination of strong operations with a majority of correct processes and at the same time to achieve Eventual Consensus and termination of weak operations with a minority of correct processes.

Interestingly, at the heart of circumventing the impossibility lies the ability to eventually tell if consensus will terminate, which is possible with $\Diamond\mathcal{P}$ but impossible with $\Diamond\mathcal{S}$. This seems to be a fundamental and unexplored difference between the two classes of failure detectors. On the other hand, a strongly complete failure detector is sufficient to eventually detect that consensus will *not* terminate.

1.2.4 Publications Resulting from the Thesis

The work reported in this thesis is supported by several international conference publications:

- **Marco Serafini**, Dan Dobre, Matthias Majuntke, Péter Bokor and Neeraj Suri, *Eventually Linearizable Shared Objects*, in Proceedings of the 29th Annual ACM SIGACT-SIGOPS Syposium on Principles of Distributed Computing (PODC), Zürich (CH), 2010.

- **Marco Serafini**, Péter Bokor, Dan Dobre, Matthias Majuntke and Neeraj Suri, *Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas*, in Proceedings of the 40th IEEE International Conference on Dependable Systems and Networks (DSN-DCCS), Chicago (US), 2010.

- **Marco Serafini** and Neeraj Suri, *Reducing the Costs of Large-Scale BFT Replication*, in Proceedings of Large-Scale Distributed Systems and Middleware (LADIS), White Plains (US), 2008.

- **Marco Serafini** and Neeraj Suri, *The Fail-Heterogeneous Architectural Model*, in Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS), Beijing (China), pp. 103–113, 2007

The following publications in international conferences and journals were developed in the context of the PhD work of this thesis. These publications span different topics, such as membership in synchronous embedded systems to and formal verification of distributed algorithms.

Applications of eventually linearizable shared objects

- **Marco Serafini** and Flavio Junqueira, *Weak Consistency as Last Resort*, in Proceedings of the 4th ACM SIGOPS/SIGACT Workshop on Large Scale Distributed Systems and Middleware (LADIS), Zürich (CH), 2010.

Membership algorithms for transient faults in synchronous systems

- **Marco Serafini**, Péter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstätter, Fulvio Tagliabó and Jens Koch, *Application-Level Diagnostic and Membership Protocols for Generic Time-Triggered Systems*, IEEE Transactions on Dependable and Secure Computing (IEEE TDSC) – accepted, to appear

- Kohei Sakurai, Masahiro Matsubara, **Marco Serafini** and Neeraj Suri, "Dependable and Cost-Effective Architecture for X-by-Wire Systems with Membership Middleware", Proc. of FISITA World Automotive Congress, 2008.

- **Marco Serafini**, Andrea Bondavalli and Neeraj Suri, *On-Line Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters*, IEEE Transactions of Dependable and Secure Computing (IEEE TDSC), 4(4), pp. 295–312, Oct. 2007

- **Marco Serafini**, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstätter, Fulvio Tagliabó and Jens Koch, *A Tunable Add-On Diagnostic Protocol for Time-Triggered Systems*, in Proceedings of the 37th IEEE International Conference on Dependable Systems and Networks (DSN-DCCS), Edinburgh (UK), pp. 164–174, 2007

Model checking of distributed algorithms

- Peter Bokor, **Marco Serafini** and Neeraj Suri, "Efficient Models for Model Checking Message-Passing Distributed Protocols", Proc. of Formal Techniques for Networked and Distributed Systems (FORTE), 2010.

- Peter Bokor, **Marco Serafini** and Neeraj Suri, "Role-Based Reduction of Fault-Tolerant Distributed Protocols with Language Support", Proc. of Int'l Conf. on Formal Engineering Methods (ICFEM), 2009.

- Peter Bokor, **Marco Serafini**, Helmut Veith and Neeraj Suri, "Efficient Model Checking of Fault-tolerant Distributed Protocols Using Symmetry Reduction (Brief Announcement)", Proc. Int'l Symp. on Distributed Computing (DISC), 2009.
- Peter Bokor, **Marco Serafini**, Aron Sisak, Andras Pataricza and Neeraj Suri, "Sustaining Property Verification of Synchronous Dependable Protocols Over Implementation", Proc. of the IEEE Int'l Symp. on High Assurance Systems Engineering (HASE), 2007.

Byzantine-fault tolerant storage

- Matthias Majuntke, Dan Dobre, **Marco Serafini** and Neeraj Suri, "Abortable Fork-Linearizable Storage", Proc. of Int'l Conf. on Principles of Distributed Systems (OPODIS), 2009.
- Dan Dobre, Matthias Majuntke, **Marco Serafini** and Neeraj Suri, "Efficient Robust Storage using Secret Tokens", Proc. of Int'l Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS), 2009.

Crash-tolerant consensus over Wide Area Networks

- Dan Dobre, Matthias Majuntke, **Marco Serafini** and Neeraj Suri, "HP: Hybrid Paxos for WANs", Proc. European Dependable Computing Conference (EDCC), 2010.

1.3 Thesis Structure

The structure of the following chapters follows the structure of the research questions described earlier:

Chapter 1 presents the background of the problems driving this research, introduces the research problems and the contributions of this thesis.

Chapter 2 introduces the terminology used throughout the thesis and surveys the state of the art in fault-tolerant replication, with a particular interest for its application for Web-scale systems.

Chapter 3 describes the Scrooge protocol.

Chapter 4 defines the fail-heterogeneous fault model and introduces the HereTrust protocol.

Chapter 5 introduces eventual linearizability, shows inherent tradeoffs in implementing it, and describes the gracefully-degrading Aurora protocol.

Chapter 6 finally concludes the thesis, re-evaluating the value of the conceptual and experimental contributions. A discussion on the applicability of the thesis results to different fields of distributed systems, especially on a Web-application scale, alongside with an outline of the future research directions opened by the novel approach presented by this thesis.

Chapter 2

State of the Art and Background

Fault-tolerant replication over a message-passing distributed system is a long-established problem that spurred a large volume of research over the last decades. This chapter reviews some basic concepts of fault-tolerant replication, which are necessary to understand the contributions of this thesis. It then makes an overview of the two specific topics treated in this thesis: Byzantine-fault tolerant replication and weakly consistent replication.

2.1 The Consensus Problem and Replication

Consensus is a fundamental problem in distributed computing. It requires a set of processes starting with possibly different initial values to eventually output a single common output value. Consensus is a paradigmatic problem in distributed coordination and has been extensively studied over the last decades.

The problem of fault-tolerant consensus over message-passing distributed systems was first introduced by Lamport, Pease and Shostak in the early eighties [PSL80; LSP82a]. Byzantine-fault tolerant consensus algorithms where initially used to implement clock synchronization in avionic systems [WLG$^+$78]. In these real-time dedicated systems, it is safe to assume that the message-passing system is *synchronous*, that is, there exists a known upper bound on the message communication and processing delay of each process. The initial work of Lamport, Pease and Shostak also established the lower bound on the number of replicas necessary to tolerate a given number of Byzantine faults using a synchronous message-passing system. The lower bound on the time complexity, expressed in terms of number of communication rounds, followed shortly thereafter [FL81].

Subsequent research examined the problem of consensus in different classes of systems where communication may be asynchronous, or partially synchronous, but only crashes are tolerated. In the crash fault model, processes follow their specification until they stop taking any step, and messages can not be corrupted. An early, fundamental result was the impossibility of solving consensus in *asynchronous* systems, where there is no upper bound on message communication and processing delays [FLP85]. A palette of different *partial synchrony* models representing the minimal synchrony conditions to solve consensus was proposed in [DDS87].

2.1.1 Failure Detectors

Partial synchrony can be expressed by augmenting the asynchronous system model with the abstraction of *failure detectors* [CT96]. Failure detectors are oracles providing information on which processes have crashed. Each process runs a failure detector module that outputs at any time a set of process indices. Failure detectors are grouped in different classes based on their completeness and accuracy. Completeness refers to the ability of a failure detector to eventually suspect all crashed processes. Accuracy requires that correct processes are not suspected. Partially synchronous systems can be modeled as systems with an eventually accurate failure detector, which can mistakenly suspect all correct processes as faulty for a finite time. Since suspects of these failure detectors are unreliable, consensus algorithms need to be *indulgent* and deal with false suspicions [Gue00].

Failure detectors represent a way to express the inherent complexity of solving a distributed computing problem. A good survey on the failure detector abstraction is [FGK06]. Much work has dealt with identifying the weakest failure detectors that are necessary to solve distributed computing problems, as for example [CHT96; DGFG$^+$04]. In this thesis we consider four classes of failure detectors. The class Ω is the weakest class of failure detector to solve consensus. Failure detectors of class Ω output at most one process id at each process p_i. The process whose id is output is said to be *trusted* by p_i. All failure detectors of class Ω eventually let a single correct process be permanently trusted by all correct processes [CHT96]. The class of *strongly complete* failure detectors, which we denote $\mathcal{C}$, includes all failure detectors that output a set of *suspected* processes and that ensure *strong completeness*, i.e., eventually every process that crashes is permanently suspected by every correct process [CT96]. The classes of *eventually strong* (resp. *eventually perfect*) failure detectors $\Diamond\mathcal{S}$ (resp. $\Diamond\mathcal{P}$) include all strongly complete failure detectors having *eventually weak accuracy* (resp. *eventually strong accuracy*), i.e., eventually some correct process is (resp. all correct processes are) not suspected by any correct process [CT96].

2.1.2 The Paxos Protocol

Paxos is a very simple and efficient algorithm to solve consensus in the crash model using a leader oracle [Lam98; Lam01]. It identifies three roles for processes. *Proposers* have an initial value and propose it to become the final output value. They send their proposals only when the leader oracle indicates them as leader. *Acceptors* accept proposals. If enough acceptors has accepted a proposal, this is termed as *chosen* and can be safely learned as output value by learners. *Learners* establish that a proposal can be decided as output.

The communication pattern of Paxos in "good runs" where there is only one leader proposer is depicted in Figure 2.1 Before making a proposal, a leader reads from the acceptors to find out if any previous proposed value may have been learned. If such a value is found, the leader adopts it as its own initial value. In this step, acceptors promise to the leader that they will ignore all messages sent by any other previous leader. In order to establish a total order between the leaders, a *proposal number* is associated to each message sent by the leader. Whenever a process is elected as a leader, it increases its proposal number. Proposal numbers are unique: no two different processes ever use the same proposal number.

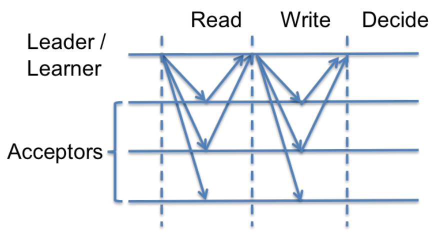

Figure 2.1: Communication pattern of the Paxos protocol, described using the terminology of [Lam01]. For simplicity, we depict the leader process as the only learner.

In the second round, the leader sends its proposal to all acceptors. If an acceptor accepts the proposal (because it has not previously promised to ignore it) it sends an acknowledgement to the leader. If enough acceptors have accepted a proposed value, learners can decide to output it.

Paxos requires $2t + 1$ processes to tolerate t crashes, which is shown to be minimal in [CT96]. The following is an informal explanation of why this number of replicas is necessary. Consensus requires that if a learner has decided a value, no other learner will decide on a different value. If at most t processes can fail by crashing, a process can wait for at most $n - t$ messages at each round. This is a consequence of the unreliability of failure detection, which makes it impossible to determine with certainty whether the sender of the t missing messages are faulty or simply slow. A learner must thus be able to learn a value after receiving $n - t$ acknowledgements. If a new leader is elected, it must be able to read the chosen value by contacting $n - t$ acceptors in the read phase. This is key for safety. It is thus easy to see that having at least $2t + 1$ replicas ensures that any two sets of acceptors having cardinality $n - t$ intersect in at least one acceptor, which then reports to the new leader the chosen value.

2.1.3 State-Machine Replication

Replicating functionalities over multiple physical devices for fault tolerance is a common technique in systems design. It is used at different layers of abstraction, from hardware design to software applications. A fundamental fault-tolerant replication technique is the state-machine approach [Sch90]. State machines model deterministic servers. They atomically execute commands issued by clients. This results in a modification of the internal state of the state machine and/or in the production of an output to a client. An execution of a state machine is completely determined by the sequence of commands it executes, and is independent of external inputs such as timeouts.

A fault-tolerant state machine can be implemented by replicating it over multiple processors. Commands need to be executed by every replica in a consistent order, despite the fact that different replicas might receive them in different orders. In state machine replication, consensus, or the equivalent *atomic broadcast* primitive [CT96], can be used by replicas to agree on a single execution order. In this case, an instance of consensus is executed to agree on the command corresponding to each sequence number in the execution order.

The Paxos algorithm uses consensus as fundamental building block to implement state machine replication. A key aspect for the efficiency of Paxos in this case is that a new leader can execute the read phase only once for all instances of consensus, or equivalently, for all sequence numbers. Therefore, the actual processing overhead for each command when there is a stable leader consists only of executing the write phase.

The correctness property implemented by state machine replication is Linearizability [HW90], which requires that clients observe commands from other clients in a total order that is consistent with the real-time order of these commands. Linearizability implicitly identifies a single "linearization point" in time where a command takes effect. This must be enclosed between the times of invocation and of completion of a command.

2.2 Modern Byzantine-Fault Tolerance

As already discussed, initial work on Byzantine-Fault Tolerance (BFT) focused on synchronous systems. Algorithms for asynchronous Byzantine agreement where already proposed in the eighties [BT85]. However, these are randomized algorithms with a highly variable performance overhead. A more recent revamp of interest on BFT started with Malkhi's and Reiter's work on Byzantine-fault tolerant quorum system, which were advocated as a method to tolerate worst-case failures in storage [MR97]. Interest in the Byzantine fault model became particularly strong after the work on the PBFT algorithm, which showed that state machine replication in partially synchronous systems can be efficient and have stable performance [CL99].

2.2.1 The PBFT algorithm

The PBFT algorithm shares some commonalities with Paxos, such as the use of a single leader to order operations, but tolerates Byzantine faults.

Some work, as for example [CC10; LAC07; RMS10], attempted to provide an unified view of the two algorithms.

Despite many commonalities, there are also important differences between Paxos and PBFT. First, Byzantine leaders need to be tolerated. The write phase of PBFT has three communication steps. The first two steps ensure that all replicas receive the same order of operations from the leader even if this is Byzantine. These two steps constitute an instance of consistent broadcast, which is similar to the echoing techniques introduced in [BT83]. The third communication step of the write phase guarantees that enough replicas have observed the order of a given operation and that this order can be recovered. After this third step is completed, an operation can be committed and completed. The communication pattern of read phase of PBFT is also more complex because the new leader needs to convince the acceptors that the value it proposes has been correctly determined by executing the read phase.

The second difference between PBFT and Paxos is that Byzantine acceptors need to be tolerated. A correct leader needs to take into account the presence of Byzantine acceptors that might lie about the values they have accepted from previous leaders. This results in a much more complex procedure used by new leaders to chose their proposed value [Cas01].

Finally, PBFT uses an eventual synchrony model requiring that eventually all messages sent among correct processes are received before receivers timeout. The reason is that designing failure detectors for Byzantine faults is still an unsolved problem because, unlike crashes, the semantic of these faults is specific to the particular application or protocol being executed (see [FGK06] for a survey).

Note that none of these problems can be solved simply by using cryptographic techniques. Byzantine processes, in fact, can disrupt the algorithm in more subtle ways than by forging messages. For example, they can pretend that they did not send or receive some message. However, using public-key cryptography does simplify aspects of the design of BFT algorithms, although at the cost of a high computational overhead. A key advantage for the efficiency of the PBFT algorithm is its exclusive use of symmetric-key cryptography, different from previous algorithms such as [KMMS98] that heavily rely on digital signatures.

PBFT uses a minimal number of replicas, matching the lower bound shown in [BT85]. In fact, it requires $3f + 1$ processes to tolerate f Byzantine faults.

	Replication costs (min. $2f + b + 1$ [Lam03])	Fast w. no unresponsive replicas	Fast w. f unresponsive replicas
PBFT [CL99]	$3f + 1$	NO	NO
Zyzzyva [KAD$^+$07]	$3f + 1$	YES	NO
Zyzzyva5 [KAD$^+$07]	$5f + 1$	YES	YES
DGV [DGV04]	$3f + 2b - 1$	YES	YES
Scrooge	$2f + 2b$	YES	YES

Table 2.1: Comparison of primary-based BFT replication protocols that tolerate f failures, including b Byzantine ones.

2.2.2 Efficient Byzantine-Fault Tolerance

Many algorithms subsequent to PBFT tried to improve its efficiency, in particular by reducing the number of steps executed when there exists a stable leader. Multiple results showed an implicit tradeoff between latency and replication costs, and introduced matching algorithms such as Fab and DGV [MA06; DGV04]. These fast algorithms merge the last two steps of the write phase into a single one one. As a result, processing a command in presence of a stable leader requires only two steps instead of three. The resulting communication pattern for the write phase is similar to the one of Paxos in Figure 2.1, with the difference that there are multiple learners receiving the message in the last step.

The Zyzzyva protocol further optimized these algorithms by letting clients, rather than replicas act as learners. This reduces the latency in presence of a stable leader to only three steps: from client to the leader, from the leader to the other replicas, and from these to the client. This leads to important advantages in terms of latency and throughput, but still must adhere to the implicit tradeoffs between being fast and having minimal replication costs identified in [MA06; DGV04].

A comparison of Scrooge with PBFT [CL99], Zyzzyva [KAD$^+$07] and DGV [DGV04] is already dicussed in chapter 1, and is summarized again in Table 2.2.

In [GKQV10] a framework is proposed where different protocols can be combined to react to different systems conditions. The authors present two new protocols which improve the latency or the throughput of BFT replication in fault-free runs where specific preconditions are met (e.g. clients do not submit requests concurrently). In presence of unresponsive replicas, these protocols need to switch to a backup protocol such as PBFT.

Protocols like Q/U [AEMGG$^+$05] and HQ [CML$^+$06] let clients directly interact with the replicas to establish an execution order. This reduces latency in some cases but is more expensive in terms of of MACs operations [KAD+07; SDM+08].

Preferred quorums is an optimization used by clients in some quorum-based BFT replication protocols to reduce the cryptographic overhead or to keep persistent data of previous operations of the client [CML+06; AEMGG+05]. Preferred quorums are not agreed-upon using reconfigurations and are not used during view change. This technique is thus fundamentally different from replier quorums because using (or not using) it has no effect on the replication cost of the protocol.

2.2.3 Trusted Components and Confidentiality

Byzantine agreement protocols and the homogeneous fail-Byzantine model were introduced to tolerate arbitrary physical faults in synchronous safety-critical systems [LSP82b]. Distributed systems designed to tolerate f Byzantine faults can in general handle $m \geq f$ less severe faults, although not necessarily at the same time. In order to model this, hybrid fault models [Lam03; MP91; WS03] assume that *any* node in the system can fail in a malicious or benign manner, as long as the upper bound on the number of faulty nodes is respected.

Hybrid architectures partition the system into different subsystems with different sets of assumptions. For example, the Wormhole model [Ver06] considers systems that are partitioned into multiple subsystems, which can be characterized by different failure modes and synchrony assumptions. An example of architecture built conforming to this model is TTCB [CVN02], where each node is composed of two different subsystems. The first is an asynchronous, fail-Byzantine payload subsystem connected to the payload subsystems of the other nodes through an asynchronous payload channel. The second is a synchronous, fail-crash control subsystem with limited computational capabilities and usually connected to the other control subsystems in other nodes through a dedicated, low bandwidth and synchronous control channel.

The fail-heterogeneous architectural model differs from hybrid fault models as it associates different fault models to specific nodes of the distributed system. It also differs from the Wormhole model as it does not consider different subsystems internal to nodes, nor different degrees of synchrony within specific subsystems.

The BFT protocol [CL99] for homogeneous fail-Byzantine systems implements state machine replication and guarantees that replicas do not diverge even in presence of malicious attacks and intruded participants. Compared to the Paxos protocol [Lam98; Lam01], which is its fail-crash counterpart, BFT requires more replicas to tolerate f faults ($3f+1$ instead of $2f+1$) and has a higher latency. Subsequent work showed that a latency comparable to the crash-only case is achievable at the cost of a higher degree of replication ($5f+1$) [MA06].

If agreement and execution are separated, as proposed in [YMV$^+$03], agreement processes can have a simple design and require fewer local resources, while only $2f+1$ complex replicas of the servers need to be diversified (using a proper abstraction layer such as [RCL01]). However, to keep the number of faulty processes below the upper bound f there should be no correlation between failures, i.e., intrusions, at any different replica. As intrusions are made possible by design faults, such as vulnerabilities, failure independence requires diversified design of each *node* participating in the protocol (e.g., different operating systems must be used, different applications etc.) regardless of its role.

Based on similar considerations, our HeterTrust protocol assumes the availability of a set of simple nodes dedicated to replica coordination. Since the coordination algorithm is generic and can be re-used in multiple contexts, a thorough verification and validation of its design can be worthy.

As pointed out in [YMV$^+$03], replication of confidential data increases the likelihood that an attacker can intrude a replicated server and obtains confidential information. The authors of [YMV$^+$03] propose a privacy firewall to make sure that (a) only replies processed by at least one correct process might be sent out by the service, and (b) replies should be as deterministic as possible to prevent attackers from using steganography. This represents the best solution proposed so far under a fail-Byzantine model. However, it requires a high number of replicas, a long latency to filter replies, and expensive threshold cryptography to make replies deterministic. The fail-heterogeneous architecture of HeterTrust represents a viable alternative to achieve the same properties with lower overhead and fewer replicas.

HeterTrust uses a majority of correct fail-crash coordination nodes to reduce the number of complex fail-Byzantine execution nodes with diversified design to $2f+1$. If trusted coordinators are incorporated as sub-components of execution servers, the HeterTrust only needs a majority of correct processes. In [CNV04], a similar result was achieved by relying on the synchrony of TTCB communication for agreement. HeterTrust tolerates periods of asynchrony and requires only an Ω leader election protocol for liveness.

In [MBTPV06] agreement and execution are also separated in fail-crash systems to take advantage of regions with "early" partial synchrony, where reaching agreement is easier. A hierarchical protocol decomposition approach for WANs is proposed in [ACKL07]. It allows choosing different combinations

Protocol	**SM**	**FM**	n	**Latency**	**Confid.**	**Crypt.**
Paxos [Lam98; Lam01]	PS	C	$2g+1$	4/5	-	-
BFT [CL99]	PS	B	$3f+1$	4	no	MAC
FaB [MA06]	PS	B	$5f+1$	3	no	MAC
Correia *et.al* [CNV04]	W	W	$2m+1$	5	no	MAC
Marchetti *et.al* [MBTPV06]	PS	C	$3g+2$	4/5	-	-
Yin *et.al* [YMV+03]	PS	B	f^2+6f+2	$2f+7$	yes	TS
HeterTrust	PS	H	$2f+2g+2$	4	yes	MAC

n = lower bound on # nodes; $g/f/m$ = upper bound on # fail-crash / Byzantine / mixed nodes
SM = System Model (**P**artially **S**ynchronous / **W**ormhole)
FM = Fault Model (**C**rash / **B**yzantine / **W**ormhole / **H**eterogeneous)
MAC = Message Authentication Codes; TS = Threshold Signatures

Table 2.2: HeterTrust - comparison with other deterministic state machine replication protocols

of fault tolerance protocols at each site and among sites in a customizable manner to mask Byzantine faults. Our approach differs from this as it binds failure modes to specific nodes based on their design.

Table 2.2 presents a comparison between HeterTrust and other deterministic state machine replication protocols. Most of the compared protocols assume partially synchronous system models similar to [DLS88], except [CNV04] where a Wormhole model is assumed. We report the upper bounds on crash (g) and Byzantine (f) faults tolerated. In general, only a subset of the nodes required for agreement (n) needs to actually implement the replicated service (e). Multi-tier architectures require additional nodes (a), for confidentiality or for faster agreement. In this case g and f represent upper bounds for each layer. The state machine message complexity and latency is measured during best-case runs as the number of communication steps on the critical path from a client request to its reply. Where indicated by the authors, we consider the use of tentative executions [KPA+03]. For confidentiality, additional communication steps are necessary in [YMV+03] and HeterTrust. A simple variation of HeterTrust exists that does not provide confidentiality and allows saving one communication step. Table 2.2 also indicates the type of cryptography used during normal operation in the critical path. All mentioned fail-Byzantine protocols use public keys during recovery.

2.3 Weak Consistency Semantics

The previous sections have illustrated that Paxos can only make progress, when a correct leader is trusted by all correct processes and is able to communicate with $n-t$ correct acceptors. If these conditions are not met, due to asynchrony in the network, process failures or partitions, then Paxos and other algorithm implementing consensus block.[1] This is unacceptable for some applications. Also, the latency of solving consensus might be too high for some applications. These are the reasons motivating relaxations of consistency in replicated systems.

The following previous work has studied how to extend Linearizability with weaker consistency properties. Eventual Serializability requires that "strict" operations and all operations preceding them be totally ordered at the time of their response, while other operations may only be totally ordered *after* their response [FGL$^+$96]. Most existing systems implementing optimistic replication provide variations of this property, often called Eventual Consistency [SS05; Vog09]. As we show, Eventual Consistency is weaker than Eventual Linearizability. Timed Consistency strengthens sequential consistency by setting a real-time bound Δ after which operations must be seen by any other process [TRAR99]. If $\Delta = 0$ the specification is equivalent to Linearizability. If not, Timed Consistency allows completed operation to remain invisible to subsequent operations, similar to Eventual Consistency. In this case, our result can be easily extended to show that Timed Consistency is not stronger than Eventual Linearizability. Like Eventual Serializability, Hybrid Consistency requires strong operations to be linearizable with each other but relaxes the ordering between pairs of weak operations [AF92]. None of these papers discusses how to strengthen consistency in "good periods" where Consensus may be solved.

A few replication algorithms have some similarity with Aurora because they seem to implement Eventual Consensus in some "good" runs, although this property is not explicitly stated. Zeno extends Byzantine-fault tolerant state machine replication to guarantee availability and Eventual Consistency for weak operations in presence of partitions [SFK$^+$09]. Zeno appears to achieve Eventual Consensus in some "good" runs. However, Zeno relaxes Linearizability for strong operations. In fact, processes invoking weak operations are allowed to observe concurrent strong operations in different orders. The COReL algorithm is a total order algorithm using partitionable group membership, rather than failure detectors, as underlying building block [KD96]. It optimistically outputs operations before their total order is known. In runs where all correct processes are eventually in the same partition, it ensures that eventually the optimistic order of the operations is equal to the definitive, total order.

[1]Note that we refer to deterministic consensus algorithms. Randomized algorithms can achieve progress with asynchronous links but only solve consensus with probability converging to 1.

A number of distributed systems, including modern highly-available data center services such as Amazon's Dynamo [DHJ^{+}07], the Google File System [GGL03] and Yahoo's PNUTS [CRS^{+}08] allow trading Linearizability for availability in presence of partitions, which occur between geographically remote data centers as well as inside data centers [Vog09]. A survey on many practical weakly consistent systems is provided in [SS05]. A drawback of weakly consistent systems is that they are notoriously hard to program and to understand [BCvR09]. Authors of [AAL^{+}08] argue, with motivations similar to ours, that many systems aim at being "usually consistent". They propose a quantitative measure, called consistability, to study the tradeoffs between performance, fault-tolerance and consistency.

There is a large body of work on weak consistency semantics for distributed shared memories having read/write semantics. For a survey we refer to [RS95]. Eventual Linearizability is an eventual safety property that can be combined with any of these safety properties. For example, Aurora has a causal consistency property that allows implementing causal memories [Lam78]. Refined specifications of graceful degradation and corresponding implementations for transactions taking snapshots of the state of multiple objects are presented in [ZPR^{+}07]. Authors of [AT10] study graceful degradation of liveness, rather than consistency. The stronger the properties of the failure detector used by the protocol, the stronger the liveness guarantees offered to clients.

2.4 Chapter Summary

This chapter has informally overviewed some fundamental concepts and algorithms of fault-tolerant replication. It has also discussed the state of the art in the fields where this thesis contributes to allow a comparison with the contributions of the thesis, which are described in the previous chapter 1.

Chapter 3

Fast BFT at Low Cost

BFT incurs a fundamental trade-off between being *fast* (i.e. optimal latency) and achieving optimal resilience (i.e. $2f+b+1$ replicas, where f is the bound on failures and b the bound on Byzantine failures [BT85; Lam03]). Achieving fast Byzantine replication despite f failures requires at least $f+b-2$ additional replicas [MA06; DGV04; KAD$^+$07]. This chapter shows, perhaps surprisingly, that fast Byzantine agreement despite f failures is practically attainable using only $b-1$ additional replicas, which is independent of the number of crashes tolerated. This makes Scrooge particularly appealing for systems that must tolerate many crashes (large f) and few Byzantine faults (small b). The first core principle of Scrooge is to have replicas agree on a quorum of responsive replicas before agreeing on requests. This is key to circumventing the resilience lower bound of fast Byzantine agreement [DGV04]. The second is the use of the cryptographic information contained in message authenticators to detect faulty replicas.

3.1 Technical Highlights

This chapter describes the Scrooge protocol. Before introducing the details of the algorithm, however, this section gives an overview of the two main novel techniques used by the algorithm to improve on the state of the art: *replier quorums* and *message histories*.

3.1.1 First Technique: Replier Quorums

Scrooge uses two novel techniques, *replier quorums* and *message histories*, to reduce replication costs. The first technique consists of having replicas agree on a set of replicas, termed *replier quorum*, whose members are the only ones responsible for sending replies to clients in normal runs. A distinguished replica, called the primary, sends messages to the other replicas that dictate the order of execution of requests. Scrooge uses speculation so replicas directly reply to the client without reaching agreement on the execution order first (Figure 3.1.a). This allows clients to immediately deliver a reply if all the repliers are responsive and correct. If a replier becomes unresponsive or starts behaving incorrectly, this is indicated by clients to the replicas, which then execute a *reconfiguration* to a new replier quorum excluding the suspected replica.

During reconfigurations, explicit agreement is performed by the replicas (Figure 3.1.b). This is similar to PBFT, but the agreed-upon value contains two types of information: the execution order of client requests *and* the new replier quorum. Agreeing on the order of requests ensures that all client requests can complete even in presence of faulty or unresponsive repliers. Agreeing on the new replier quorum allows future requests to be efficiently completed using speculation. Coupling these agreements reduces the overhead incurred by reconfiguration. The goal of the first explicit agreement in Figure 3.1.b is just completing the ongoing request from client i. When the request of client j is received, the primary has a chance to propose a new replier quorum and let all replicas explicitly agree on it. Speculation is re-established as soon as this agreement is reached.

Scrooge requires clients to participate in the selection of repliers. Giving more responsibility to clients is common in many BFT replication protocols, such as Q/U, HQ and Zyzzyva. This is reasonable as clients are ultimately entrusted not to corrupt the state of the replicated state machine with their requests. Scrooge protects the system from Byzantine clients and ensures that they can not make replica states diverge. However, Byzantine clients can reduce the performance of the system by forcing it to perform frequent reconfigurations and to use the communication pattern of PBFT (like in the request of client j in Figure 3.1.b), which anyway allows achieving good performance, instead of using speculation (like in Figure 3.1.a). This kind of client attacks can be easily addressed by simple heuristics, for example by bounding the number of accusations a client can send in a given unit of time. This reliance on clients to indicate suspected replicas results from the use of speculation. Replicas can not observe if other replicas prevent fast agreement by not sending correct speculative replies.

Reconfigurations are avoided in existing speculative protocols such as Zyzzyva5 by using more replicas than Scrooge. Replier quorums allow reducing the replication costs to $4f + 1$ replicas when $f = b$.

3.1.2 Second Technique: Message Histories

Scrooge leverages the Message Authentication Codes (MACs) used in BFT replication protocols to implement authenticated channels and to detect forged and corrupted messages. The sender of a message generates an *authenticator*, which is a vector of MACs with one entry for each other receiver, and attaches it to the message. In current primary-based protocols such as [CL99; KAD$^+$07], replicas store the history of operations dictated by the primary but discard the authenticator of the messages from the primary after their authenticity has been verified. Scrooge lets replicas store the entire content of these messages, including the authenticator, in their *message histories*. This further reduces the replication cost from $4f + 1$ to $4f$ (again with $f = b$).

3.2 System and Fault Model

The system is composed of a finite set of clients and replicas. At most f replicas can be faulty, out of which at most b can be Byzantine (with $0 < b \leq f$) while the others can only crash. The system has $N \geq 2f + 2b$ replicas. Any number of clients can be Byzantine. Clients and replicas are connected via an unreliable asynchronous network. The network has *timely* periods when all messages sent among correct nodes are delivered within a bounded delay.

We assume the availability of computationally secure symmetric key cryptography, to calculate MACs, and public key cryptography, to sign messages. If message m is sent by process i to process j and is authenticated using simple MACs, this is denoted as $\langle m \rangle_{\mu_{i,j}}$. In case m is sent to all replicas by process i, an authenticator consisting of a vector of MACs with one entry per replica is sent with m and this is denoted as $\langle m \rangle_{\mu_i}$. If m is signed by i

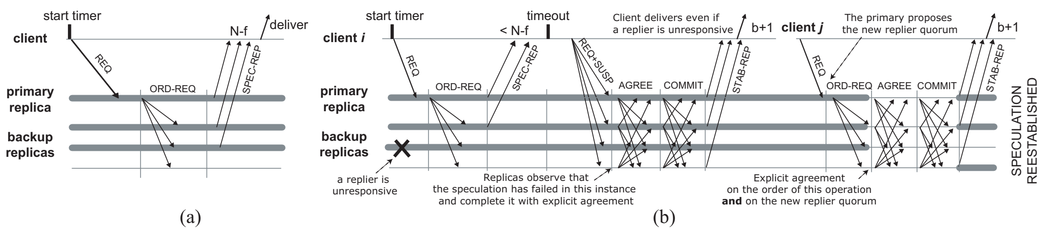
start timer
client
REQ
primary replica
ORD-REQ
backup replicas
N-f
deliver
SPEC-REP
(a)
start timer
client i
REQ
primary replica
backup replicas
a replier is unresponsive
ORD-REQ
SPEC-REP
< N-f
timeout
REQ+SUSP
AGREE
COMMIT
Replicas observe that the speculation has failed in this instance and complete it with explicit agreement
Client delivers even if a replier is unresponsive
b+1
STAB-REP
client j
The primary proposes the new replier quorum
REQ
ORD-REQ
AGREE
COMMIT
STAB-REP
b+1
Explicit agreement on the order of this operation and on the new replier quorum
SPECULATION REESTABLISHED
(b)

Figure 3.1: Communication patterns: (a) with speculation, during normal periods; (b) with explicit agreement, during transient reconfiguration periods where two client requests are processed. Repliers are indicated with a thicker line.

Name	Description	Type
v	current view	timestamp
RQ	replier quorum	set of pids
n	current seq. number	timestamp
mh	message history	array of $\langle$*req.*, RQ, *auth.*$\rangle$
h	history digests	array of digests
aw	agreed watermark	timestamp
cw	commit watermark	timestamp
SL	suspect list	set of f pids
v'	new view	timestamp
ih	initial history	array of $\langle m, RQ, auth.\rangle$
E	view establishment certificate	set of $N-f$ signed EST-VIEW messages

Table 3.1: Global Variables of a Replica

using its private key, this is denoted as $\langle m\rangle_{\sigma_i}$. Scrooge also assumes the availability of a collision-resistant hash function H ensuring that for any value m is impossible, given $H(m)$, to find a value $m' \neq m$ such that $H(m) = H(m')$.

3.3 The Scrooge Protocol

Scrooge replicates deterministic applications, modeled as state machines, over multiple servers. Clients use Scrooge to interact with the replicated servers as if they were interacting with a single reliable server. Beyond the classic safety and liveness properties necessary for BFT replication, in Scrooge clients eventually complete all their requests from speculative replies if the basic conditions for speculation are satisfied (i.e. the primary is fault-free, the clients are non-Byzantine and the system is timely).

For easier understanding, this thesis presents a simplified version of Scrooge which assumes that replicas process unbounded histories. A complete description of the full Scrooge protocol with garbage collection, together with full correctness proofs, can be found in Appendix A.

Algorithm 1: Scrooge - Normal Execution

```
1.1  upon client invokes operation $o$
1.2      $t \leftarrow t+1$;  $SL \leftarrow \bot$;
1.3      send $m = \langle$REQ, $o$, $t$, $c$, $SL\rangle_{\sigma_c}$ to the primary;
1.4      start timer;
1.5
1.6  upon primary $p(v)$ receives REQ message $m$ from client $m.c$ or a replica
1.7      if not IN-HISTORY($m$, $mh$) then
1.8          $n \leftarrow n+1$;  $d \leftarrow H(m)$;  $RQ_p \leftarrow$ replicas $\notin SL$;
1.9          send ($\langle$ORD-REQ, $v$, $n$, $d$, $RQ_p\rangle_{\mu_p}$, $m$) to all replicas;
1.10     else if not COMMITTED($m$, $mh$, $cw$) then
1.11         update($m.SL$);
1.12         agree($m$);
1.13     else reply-cache($m.c$);
1.14
1.15 upon replica $i$ receives ORD-REQ message $orq$ from primary $p(v)$
1.16     if $i = p(v)$ or ($orq.v = v$ and $orq.n = n+1$ and $p(v) \in orq.RQ_p$) and not
         IN-HISTORY($orq.m$, $mh$) then
1.17         $n \leftarrow n+1$;  $h[n] \leftarrow H(h[n-1], mh[n])$;
1.18         $mh[n] \leftarrow \langle orq.m, orq.RQ_p, orq.\mu_p\rangle$;
1.19         $r \leftarrow$ execute($orq.m.o$);
1.20         if SPEC-RUN($i$, $orq.m$, $orq.RQ_p$, $RQ$) then
1.21             if $i \in RQ$ then
1.22                 send $\langle$SPEC-REP, $v$, $n$, $h[n]$, $RQ$, $orq.m.c$, $orq.m.t$, $r$, $i\rangle_{\mu_{p,c}}$ to client $orq.m.c$;
1.23         else
1.24             agree($orq.m$);
1.25             if $RQ \neq orq.RQ_p$ then  $RQ \leftarrow \bot$;
1.26         if AGREEMENT-STARTED($i$, $n$, $v$) then agree($orq.m$);
1.27
1.28 upon client receives SPEC-REP message $sp$ from replica $sp.i$
1.29     if $|sp.RQ| = N - f$ and client received speculative replies matching $sp$ from all replicas in
         $sp.RQ$ then
1.30         deliver ($o$, $t$, $sp.r$);   stop timer ;
1.31
```

3.3.1 Normal Execution

In normal executions where the system is timely, the primary is fault-free and the replier quorum is agreed by all replicas and contains only fault-free replicas, Scrooge behaves as illustrated in Figure 3.1.a and Algorithm 1.[1] Table 3.1 summarizes the local variables used by the replicas. Replicas use only MACs for normal runs and reconfigurations.

Scrooge runs proceed through a sequence of views. In each view v, one replica, which is called the primary and whose ID is $p(v) = v \bmod N$, is given the role of assigning a total execution order to each request before executing it. The other replicas, called backups, execute requests in the order indicated by the primary.

[1] Upon receiving a message, clients and replicas discard them if they are not well-formed, i.e., if the signatures, MACs, message digests or certificates are not consistent with their definitions. Such non well-formed messages are ignored in the pseudocode.

Clients start the protocol for an operation o with local timestamp t by sending a signed *request* message REQ to the primary. Clients then start a timer and wait for speculative replies (Lines 1.1 – 1.4). When the primary receives a request for the first time (Lines 1.6 – 1.9) it assigns it a sequence number and sends an *order request* message ORD-REQ to mandate the same assignment to all backups. The primary also stores the request in its message history together with the current replier quorum RQ_p and the authenticator μ_p of the ORD-REQ message.

When a replica receives order requests from the primary of the current view (Lines 1.15 – 1.19), it checks that its view number is the current one, that it contains the next sequence number not yet associated with a request in the message history (predicate IN-HISTORY), and that the primary has included itself in the replier quorum. If all these checks are positive, the request is executed and the fields of the ORD-REQ message are added to the message history.

Speculative runs where the pattern of Figure 3.1.a is executed are the common-case runs (Lines 1.20 – 1.22). A replica checks the predicate SPEC-RUN to distinguish speculative runs. The predicate is true unless (i) a client could not complete the request out of speculative replies and has resent its request to all replicas, including backups, or (ii) the primary has proposed a new replier quorum which has not yet been agreed upon. In speculative runs, replicas send a *speculative reply* to the client if it is a replier. Beyond the reply r, the view number v and the sequence number n associated to the client request, speculative replies contain the digest of the current history $h[n]$ and the replier quorum RQ. The former allows clients to verify that the senders of speculative replies have a consistent history; the latter to identify the replicas in the current replier quorum. If a client receives matching speculative replies from all the $N - f$ replicas in RQ, it delivers the reply (Lines 1.28 – 1.30).

3.3.2 Reconfiguration

If a replica in the replier quorum fails, the client can not complete requests out of speculative replies. The replier quorum is then reconfigured by eliminating faulty repliers to re-establish the communication pattern of Figure 3.1.a. Replicas start a full three-phase agreement similar to PBFT by calling the *agree* procedure, which takes the client request as argument (see Appendix A for the full pseudocode). An example of reconfiguration over two client requests is in Figure 3.1.b.

Completion of client requests When clients cannot deliver speculative replies before the timer expires, they double the timer, indicate the IDs of the repliers which have failed to respond and require replicas to explicitly agree on a common message history. Similar to client i in Figure 3.1.b, they do this by simply resending their requests, together with the set SL of suspect replicas, to all replicas.

When the primary receives a request which is already in its message history, it checks with the predicate COMMITTED if a three-phase agreement on the order of the request has already been completed. If not, the primary adds the suspect list provided by the client to its list of the f most-recently suspected servers SL and starts agreement (Lines 1.10 – 1.12). The backups similarly start agreement because receiving the client request invalidates SPEC-RUN. However, they need to receive the corresponding ordered request from the primary first (Lines 1.23 – 1.25). A replica i also starts an agreement phase whenever another replica previously sent it an agreement message (Line 1.26).

Replicas then execute the remaining two phases of agreement, agree and commit, to converge to a consistent history and send stable replies to the client. In each phase replicas send an *agree* or a *commit* message and wait for $N - f - 1$ matching messages from the other replicas before completing the phase. The agree and commit watermarks aw and cw mark the end of the history prefix which has been respectively agreed and committed. Similar to PBFT, all correct replicas completing the agreement phase for sequence number n' have the same message history prefix up to n'. When correct replicas complete the commit phase for n', they know that a sufficient number of correct replicas have completed agreement on the history prefix up to n' to ensure that the prefix will be recovered during view change. Replicas thus send *stable reply* messages to the client. Stable replies differ from speculative replies as they indicate that the history prefix up to the replied request can be recovered. Clients can deliver after receiving a stable reply from at least one correct replica, that is, after receiving matching stable replies from *any* set of $b + 1$ replicas. Replicas cache the replies to committed requests to respond to clients re-sending their requests (Line 1.13).

Agreement on a new replier quorum The classic three-phases agreement is also executed for all subsequent requests until a new replier quorum is agreed, as in case of the request of client j in Figure 3.1.b. The primary computes a new replier quorum RQ_p from the suspect list SL in Line 1.8. It then proposes RQ_p along with the next request which is ordered. Proposing a new replier quorum invalidates the SPEC-RUN predicate for all backups and lets them start agreement (Lines 1.23 – 1.25). Replicas register ongoing reconfigurations by setting RQ to $\perp$ until a reconfiguration is completed. They then start the successive two phases of agreement. Explicit agreement on RQ_p lets replicas converge not only on a common history but also on a new replier quorum. When a replica commits, it sets RQ to the new replier quorum proposed by the primary so that SPEC-RUN holds again for future requests and speculation is re-established. The commit on the new replier quorum ensures that it will be recovered if view changes take place.

3.4 Scrooge View Change

If backups receive requests from the clients and see that the system is not able to commit them, they start a view change to replace the current primary. In contrast to PBFT, one can only expect replicas to have explicitly agreed on a *prefix* of the request history completed by clients. Also, different from existing fast protocols allowing speculation in presence of unresponsive replicas, Scrooge uses a lower number of replicas. This thesis presents a novel view change protocol (see Algorithm 2) to achieve these challenging goals. As customary, replicas now use signed messages.

3.4.1 Communication Pattern

View change to a new view v' tries to build an initial history ih for v', which is then adopted as new message history when v' is started. When a replica initiates view change from the current view v to view v', it stops processing requests, starts a timer, and sends a *view change* message VIEW-CHANGE to all replicas (see Figure 3.2(a) and Lines 2.1 – 2.5). A view change can also be initiated when a replica receives $b + 1$ view change messages for a newer view (Lines 2.15 – 2.16).

A view change message contains the new view v' that the replica wants to establish, the old view v, its message history mh, the view establishment certificate E and the agreement watermark aw. The message history mh contains, as prefix, the initial history ih_v of v, which was stored at the end of the view change to the current view v. By induction on the correctness of the view change subprotocol for a given view, ih_v contains every operation completed by any client in the views prior to v. The view establishment certificate E contains the EST-VIEW messages received at the end of the view change to view v and proves the correctness of ih_v. The remaining suffix of mh contains the ORD-REQ messages received by i from the primary of view v. These requests need to be recovered by the view change if they have

Algorithm 2: Scrooge - View change

```
2.1  procedure view-change(nv)
2.2      stop executing request processing;
2.3      $v' \leftarrow nv$;
2.4      send $\langle$VIEW-CHANGE, $v'$, $v$, $mh$, $aw$, $E$, $i\rangle_{\sigma_i}$ to all replicas;
2.5      start timer;
2.6
2.7  upon replica i receives VIEW-CHANGE message vc from replica vc.i
2.8      if $vc.v' > v$ and not yet received a VIEW-CHANGE message vc for view $nv = vc.v'$ from
         $vc.i$ then
2.9          $k \leftarrow n' + 1 : \forall ev \in vc.E, cv.n = n'$;
2.10         while $mh[k] \neq \perp$ do
2.11             $res[k] \leftarrow$ verify($vc.v$, $k$, $vc.mh[k]$);
2.12             $k \leftarrow k + 1$;
2.13         $d \leftarrow H(vc)$;  $j \leftarrow vc.i$;  $v_j \leftarrow vc.v$;
2.14         send $\langle$CHECK, $j$, $v_j$, $d$, $res$, $i\rangle_{\sigma_i}$ to $p(vc.v')$;
2.15         if received $b + 1$ vc msgs with $vc.v' > v'$ then
2.16             view-change($vc.v'$);
2.17         if $i = p(v')$ and $vc.v' = v'$ and recover-prim() then
2.18             send $\langle$NEW-VIEW, $v'$, $VC$, $CH$, $i\rangle_{\mu_i}$ to all replicas;
2.19
2.20 upon replica i receives a CHECK message ch
2.21     if $i = p(v')$ and $ch.v_j = v'$ and recover-prim() then
2.22         send $\langle$NEW-VIEW, $v'$, $VC$, $CH$, $i\rangle_{\mu_i}$ to all replicas;
2.23
2.24 upon replica i receives a NEW-VIEW message nv
2.25     if not yet received nv with $nv.v' = v'$ from $p(v')$ and recover($nv.VC$, $nv.CH$) then
2.26         $h \leftarrow H(ih)$;
2.27         $n \leftarrow$ length($ih$);
2.28         send $\langle$EST-VIEW, $v'$, $n$, $h$, $i\rangle_{\sigma_i}$ to all replicas;
2.29
2.30 upon replica i receives an EST-VIEW message ev
2.31     if received set $E_{v'}$ of $N - f - 1$ ev msgs: $ev.v' = v'$ and $ev.h = H(ih)$ and $ev.n$ = length(ih)
         then
2.32         $mh \leftarrow ih$;  $v \leftarrow v'$;  $E \leftarrow E_{v'}$;
2.33         $aw, cw \leftarrow \max\{k : mh[k] \neq \perp\}$; $RQ \leftarrow mh[cw].RQ$;
2.34         start executing request processing;
2.35
```

been observed by any client.

A novelty of Scrooge is that each replica which receives the view change message from i checks if the messages in the history mh has been actually sent by the primary of view v (see Figure 3.2(b)). Let $vc.v$ be the value of the current view field v contained in a view change message vc sent by replica i to replica j. Scrooge executes one additional step during view change to validate that all history elements in vc, except those in the initial history of view $vc.v$, have been built from original order request messages from the primary of view $vc.v$ (Lines 2.7 – 2.14). When j receives vc, it first verifies that the new view field $vc.v'$ is higher than the current view v of j and that i has not already sent to j a view change message for the same view. Next, j checks if the elements in the message history of i are "authentic". For each element with sequence number k, j calls the *verify* function (see Algorithm 3) which first rebuilds the order request message sent by the primary of view $vc.v$ to i for sequence number k, and then verifies the authenticator of the message. Message histories make the first operation possible because they contain sufficient information to rebuild the original order request messages, including the message authenticator $\mu_{p(vc.v)}$ used by the primary of view $vc.v$. Replica j verifies the authenticator by calculating the MAC of the rebuilt order request message and by returning *true* if and only if this MAC is equal with the entry of j in $\mu_{p(vc.v)}$. The results of the verification of each element in the message history of vc is stored in a vector res, which is sent to the primary of the new view v' in a CHECK message together with additional information to associate the check message to vc.

Different from existing algorithms, the new primary only recovers from *stable* view change messages that are consistently checked by at least $b+1$ replicas (Figure 3.2(c)). If these messages claim that the message history is authentic, the history is termed as *verified*. The purpose of the additional check step will become clearer at the end of this section, when the details of recovery are discussed. For the moment, it is sufficient to note that all view change messages eventually become stable in timely periods and that the goal of this step is ensuring that if the primary of the old view v is non-Byzantine and i stores correct ORD-REQ messages in its history, then: (P1) the message history becomes verified because it receives positive CHECK messages from all correct replicas, which are at least $b+1$, and (P2) no forged, inconsistent history can receive a positive CHECK message by any correct replica and thus become verified.

The primary of a new view v' calls the *recover* function (see Algorithm 3) to try to recover the initial history ih whenever it receives a view change message (Lines 2.17 – 2.18) or a check message (Lines 2.20 – 2.22) for v'. Re-

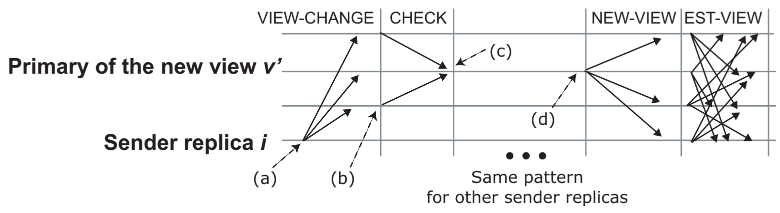

Figure 3.2: Scrooge view change subprotocol. *(a)* Replica i sends a VIEW-CHANGE message vc. *(b)* Other replicas verify the authenticator of vc. The outcome is included in a CHECK message sent to the new primary. *(c)* View change messages is *stable* when it is consistently checked by at least $f+1$ replicas. *(d)* The same is repeated for other replicas until the primary proposes an *initial history* for the new view, which is then agreed.

covery examines only stable VIEW-CHANGE message for the new view. The procedure returns *true* only if it is able to successfully recover all operations completed by any client in all views prior to v'. In this case, the resulting history forms the initial history of v' and is stored in ih. We now continue illustrating the communication pattern and argue about the correctness of the recover function in the next subsection.

If history ih is recovered, the primary sends a *new view* message to all other replicas with the sets of view change and check messages VC and CH used for the recovery (see Figure 3.2(d)). When a backup receives a new view message for the view it is trying to establish (Lines 2.24 – 2.28) it executes the same deterministic *recover* function as the primary does on the same set of view change and check messages to build the same initial history. If the backup recovers an initial history ih for a new view v', it sends an *establish view* message to all other replicas in order to agree on ih. If it later receives $N - f - 1$ establish view messages for v' consistent with ih, it forms a view establishment certificate for ih, sets v' as its current view and ih as its agreed history prefix, and updates the watermarks (Lines 2.30 – 2.34). The replica then starts processing messages in the new view.

If the replica timer expires before the new view is established, a view change to a successive new view $v' + 1$ is started, the timer is doubled and all messages related to the view change to v' are discarded.

3.4.2 The Recover Function

The *recover* function (see Algorithm 3) is a critical component because it guarantees that safety is preserved and that each history prefix observed by any correct client in previous views is also a prefix of the initial history ih of the next view. In order to allow the expert reader to verify all the nuances of the algorithm, and in particular of recovery, Table 3.2 lists the predicates used in the pseudocode.

Before starting recovery, a replica i makes sure that it has received a set VC_s of at least $N - f$ *stable* view change messages for the new view v'. A view change message vc is stable if each element in the corresponding message history $vc.mh$ is consistently verified by at least $b+1$ check messages received by the new primary (Lines 3.9 – 3.10). In timely periods each view change message vc sent by correct replicas eventually become stable as all $N - f \geq f + 2b > 2b$ correct replicas send CHECK messages containing binary vectors res for $vc.mh$.

Recovery starts by selecting an initial prefix for the initial history ih (Lines 3.11 – 3.15). The highest current view mv included in a view change message in VC_s is either the last view $lv < v'$ where some client has completed

Algorithm 3: Scrooge - View change procedures

```
function verify($v$, $n$, $e$)
    $d \leftarrow H(e.m)$; $or \leftarrow$ (ORD-REQ, $v$, $n$, $d$, $e.RQ$);
    $\mu \leftarrow$ calculate-MAC($or$, $p$);
    if $\mu = e.\mu_p[i]$ then return true;
    else return false;

function recover($VC$, $CH$)
    $recovered \leftarrow false$;
    $VC_s \leftarrow VC \setminus \{vc \in VC : \neg\text{STABLE}(vc, CH) \vee vc.v' \neq v'\}$;
    if $|VC_s| \geq N - f$ then
        $mv \leftarrow \max\{\overline{v} : \exists vc \in VC_s$ with $vc.v = \overline{v}\}$;
        $vc_{mv} \leftarrow vc \in VC_s$ with $vc.v = mv$;
        $n_{mv} \leftarrow \overline{n} : \forall ev \in vc.E, ev.n_v = \overline{n}$;
        $ih \leftarrow \{vc_{mv}.mh[k] : (k \leq n_{mv})\}$;
        $RQ_{mv} \leftarrow vc_{mv}.mh[n_{mv}].RQ$;
        $k \leftarrow n_{mv} + 1$; $loop \leftarrow true$; $recovered \leftarrow true$;
        while $loop$ do
            $A \leftarrow \{e$ : AGREED-CAND($e$, $k$, $mv$, $VC_s$, $ih$)$\}$;
            $O \leftarrow \{e$ : ORDERED-CAND($e$, $k$, $mv$, $VC_s$, $RQ_{k-1}$, $ih$)$\}$;
            if WAIT-AGR$(A, k, mv, VC_s)$ or WAIT-ORD$(A, O, k, mv, VC_s)$ then
                $loop, recovered \leftarrow false$;
            else
                if $\exists e \in A$ then
                    $ih[k] \leftarrow e$;
                else if $\exists e \in O$ : VERIFIED$(e, VC_s, CH)$ then
                    $ih[k] \leftarrow e$;
                else if $\exists e \in O$ then
                    $ih[k] \leftarrow e$;
                else $loop \leftarrow false$;
                $RQ_k \leftarrow ih[k].RQ$;
            $k \leftarrow k + 1$;
    return $recovered$;

function recover-prim()
    $VC \leftarrow$ set of received view change messages for view $v'$;
    $CH \leftarrow$ set of received check messages for view $v'$;
    return recover($VC, CH$);

```

a request, or a successive view where all requests completed in lv have been recovered. This is because at least $N - f - b \geq f + b$ correct replicas must have established lv and at least $b > 0$ of them have sent a message included in VC_s. Scrooge first recovers the initial history ih of mv from any view change message containing a message history for mv. View change messages include a view establishment certificate E composed of $N - f$ signed messages all containing the same length n_{mv} of the initial history of mv and the same corresponding history digest. The certificate ensures that the initial history ih recovered from the view change message vc_{mv} is the correct initial history for mv and is not forged by a Byzantine replica. Together with the initial history also the initial recovery quorum RQ_{mv} is recovered.

The next step is recovering the history elements observed by clients during

IN-HISTORY$(m,\, mh) \triangleq \exists k : mh[k].m.c = m.c \wedge$
    $mh[k].m.t \geq m.t$

COMMITTED$(m,\, mh,\, cw) \triangleq \exists k \leq cw : mh[k].m.c = m.c \wedge$
    $mh[k].m.t \geq m.t$

NEXT$(mh, k) \triangleq (\forall k' < k, mh[k'] \neq \bot) \wedge mh[k] = \bot$

SPEC-RUN$(i,\, m,\, RQ_p,\, RQ) \triangleq RQ \neq \bot \wedge RQ_p = RQ \wedge$
    i is backup and has never received a message
    with timestamp $\geq m.t$ from $m.c$

AGREEMENT-STARTED$(i,\, n,\, v) \triangleq$ i has received
    an agree message ag with $ag.n = n$ and $ag.v = v$ in view v

STABLE$(vc, CH) \triangleq \exists bool : \forall k : vc.mh[k] \neq \bot,$
    $\exists (b+1)\ ch \in CH : ch.v_j = vc.v \wedge ch.j = vc.i \wedge$
        $ch.d = \text{digest}(vc) \wedge ch.res[k] = bool$

AGREED-CAND$(e,\, k,\, v,\, VC,\, ih) \triangleq$
    not IN-HISTORY$(e,\, ih) \wedge$
    $\Big(\exists\ (b+1)\ vc' \in VC : vc'.v = v \wedge vc'.mh[k] = e\Big) \wedge$
    $\Big(\ \exists(|VC| - f - b)\ vc \in VC :$
        $e = vc.mh[k] \wedge vc.v = v \wedge vc.aw \geq k\ \Big)$

ORDERED-CAND$(e,\, k,\, v,\, RQ,\, VC,\, ih) \triangleq$
    not IN-HISTORY$(e,\, ih) \wedge$
    $\exists(|VC| - f - b)\ vc \in VC :$
        $e = vc.mh[k] \wedge vc.v = v \wedge vc.i \in RQ \wedge vc.aw < k$

WAIT-AGR$(A,\, k,\, v,\, VC) \triangleq \exists e \in A :$
    $\Big(\nexists\ (b+1)\ vc \in VC : vc.v = v \wedge vc.mh[k] = e)\Big) \wedge$
    $\Big(\nexists\ (f+b+1)\ vc' \in VC :$
        $(vc'.v \neq v) \vee (vc'.v = v \wedge vc'.mh[k] \neq e\Big)$

WAIT-ORD$(A,\, O,\, k,\, v,\, VC) \triangleq |A \cup O| > 1 \wedge |VC| \leq N - f \wedge$
    $(\exists\ vc \in VC : vc.i = p(v) \wedge vc.v = v)$

VERIFIED$(e,\, VC,\, CH) \triangleq \exists k,\, vc \in VC : e = vc.mh[k] \wedge$
    $(\exists(b+1)\ ch \in CH : ch.v_j = vc.v \wedge ch.j = vc.i$
    $\wedge\ ch.d = \text{digest}(vc) \wedge ch.res[k] = true)$

Table 3.2: Predicates used in the pseudocode

view mv for sequence numbers $k > n_{mv}$ (Lines 3.16 – 3.31). If a request has been completed by a client from b+1 stable replies, at least one correct replica has committed the entire history prefix up to that request (Lines 3.23 – 3.24). Committed histories are recovered like in PBFT (see predicates AGREED-CAND and WAIT-AGR). Therefore, this discussion will focus on recovering histories completed by clients through speculative replies.

Why are replier quorums useful? If a reply is delivered by clients in a *fast* manner, i.e., out of speculative replies (Lines 1.28 – 1.30), then recovering it requires a higher redundancy than the minimum. Scrooge reduces these additional costs. By recovering agreed history elements, a replica also recovers the replier quorum which has been updated when the element has been committed. Recovering the replier quorum RQ_n committed for sequence number n allows to clearly identify the set of repliers for sequence numbers greater than n and thus to reduce the number of required replicas to $2f + 2b + 1$. To see that, consider a system having $N = 2f + 2b + 1$ replicas where replier quorums consist of $N - f$ replicas. Assume that a client completes a request in a view v for sequence number $n' > n$ after receiving matching speculative replies from all repliers, at least $N - f - b$ of which are correct, and assume that RQ_n is the last recovered replier quorum for sequence numbers smaller than n'.

If the primary fails, the history prefix up to n' must be recovered to ensure safety. To this end, all replicas share their history, but only the histories of repliers in the replier quorum need to be considered. During view change up to f of the $N - f - b$ correct repliers might be slow and might fail to send a stable VIEW-CHANGE message. Due to the asynchrony of the system, the primary can not indefinitely wait for these messages because it can not distinguish if the replicas are faulty or simply slow. Despite this, the new primary can always receive view change messages from at least $N - 2f - b = b + 1$ correct repliers reporting the history prefix observed by the client. As the primary knows the identity of the repliers and as only b Byzantine repliers can report incorrect histories, the observed prefix can be recovered by selecting a history reported in the VIEW-CHANGE message of at least $b + 1$ repliers.

Why are message histories useful? Scrooge further reduces the replication costs to $N = 2f + 2b$ replicas by using message histories and the check messages. Assume that a client has delivered a reply to a request m after receiving matching speculative replies from all repliers for a sequence number n'. During view change, as Scrooge uses one replica less than the previous case, the history observed by the client is reported in the VIEW-CHANGE message of at least $N - 2f - b = b$ repliers. Let $|VC_s| \geq N - f$ be the number of stable view change messages received by the primary of the new view. We call a history element reported by $|VC_s| - f - b$ repliers an *ordered candidate*. The set of ordered candidates is defined by the predicate ORDERED-CAND. It follows from this definition that two different ordered candidates may be reported for sequence number n' and view v by two sets Q and Q' of $|VC_s| - f - b = b$ repliers each, where Q contains correct repliers and Q' the Byzantine ones. The problem is distinguishing the candidate containing m from other candidates.

If two sets of b replicas claim to have two inconsistent histories for the same view v *and* the old primary p of view v is in one of these sets, then either p is Byzantine and has sent inconsistent order requests to the backups, or b backups are Byzantine and are reporting a forged history. Therefore, at least one Byzantine replier is contained in one of these two sets and it is thus live to wait for the view change message from one additional correct replier as indicated by the predicate WAIT-ORD. After the additional VIEW-CHANGE message has been received and has become stable, $|VC_s| > N - f$. As only the correct history is reported by at least $|VC_s| - f - b > b$ repliers, it is recovered as the only remaining ordered candidate (Lines 3.27 – 3.28).

If there are two different candidates reported by b replicas each and the primary is *none* of these sets, there are two cases to consider. If p is not Byzantine, but potentially faulty, it might be impossible to wait until only one ordered candidate remains. In this case the predicate WAIT-ORD is false and a verified candidate is recovered if present (Lines 3.25 – 3.26). Message histories and the novel check phase allow to identify in these cases the history prefix observed by the client. In fact, recovery uses stable view change messages whose history elements are verified by $b + 1$ check messages in CH with consistent positive outcomes (Lines 3.9 – 3.10). Clients only deliver a speculative reply if all the repliers, including the non-Byzantine primary, have the same message history of ORD-REQ messages. This and the properties (P1) and (P2) of the check phase ensure that the history element observed by the client is verified and recovered.

The second case is when the old primary p is Byzantine. This implies that at most $b - 1$ Byzantine repliers are included in the two sets reporting the two different ordered candidates. Two correct repliers have thus received inconsistent histories from the primary. This inconsistency is detected by the client by checking the history digest of the SPEC-REP messages. Therefore the client does not deliver the reply, a contradiction.

Validity Unlike other protocols, Scrooge allows a request to be included into ih even if it is only reported by b Byzantine replicas. As client requests are signed, no request in ih is fabricated on behalf of correct clients, as commonly required for Validity by BFT replication protocols, e.g. [GKQV10].

3.5 Evaluation and Comparison

We conduct a comparative evaluation of Scrooge with other existing protocols: the standard PBFT protocol and two state-of-the-art fast protocols with publicly-available implementation, Zyzzyva and Zyzzyva5. The goal of the evaluation is to show that, during normal executions, Scrooge does not introduce significant additional overheads in the critical path compared to other speculative protocols such as Zyzzyva and Zyzzyva5. We also show that Scrooge improves over the performance of Zyzzyva in presence of unresponsive replicas, reaching the same performance as Zyzzyva5 but with less replicas. Scrooge adds two types of overhead in the critical path. First, it uses larger history elements which include authenticators. This increases the overhead of calculating the history digests included in the speculative replies. Second, speculative replies must include a bitmap representing the current replier quorum. The experimental evaluation shows that these overheads are negligible.

We refer to [SDM+08] for a comparison between quorum- and primary-based algorithms. As a reference, however, the performance figures of Q/U [AEMGG+05] scaled to the considered experimental setting are reported.

Optimizations Scrooge uses optimizations similar to PBFT and Zyzzyva to improve the performance of the protocol. The main difference between Zyzzyva and Scrooge is the read-only optimization. This lets clients send read-only requests directly to the replicas, which immediately reply to the request without having the primary order them. If this does not succeed, the client sends the read as a regular request [CL99; KAD+07]. In Scrooge, the optimization succeeds if clients receive $N - f$ consistent replies from replicas in the same replier quorum. In Zyzzyva, all replicas need to send consistent replies for the read optimization to succeed. Also, the Zyzzyva library uses a *commit optimization* to avoid excessive performance degradation with unresponsive replicas. If clients cannot receive speculative replies from all replicas, the protocol stops using speculation for successive requests and use one all-to-all agreement round instead [KAD+07].

Batching improves the performance of BFT algorithms under high load by letting replicas execute the protocol on groups of client requests [CL99]. Using batching similarly impacts all evaluated algorithms, making it more difficult to compare their performance under high load [SDM+08].

PBFT, Zyzzyva and Zyzzyva5 use MACs for client requests but this makes them vulnerable to client attacks [CWA+09]. Scrooge tolerates such attacks by using signed client requests. For fairness and consistency with previously published results, this comparison lets all algorithms use MACs.

Evaluation setup The experimental setting tolerates a single fault ($f = b = 1$). PBFT, Zyzzyva and Scrooge use four replicas while Zyzzyva5 uses six. All machines in the experiments have Intel Core2DUO 6400 2.1GHz processors, 4 GB of memory and Intel E1000 network cards, and are connected

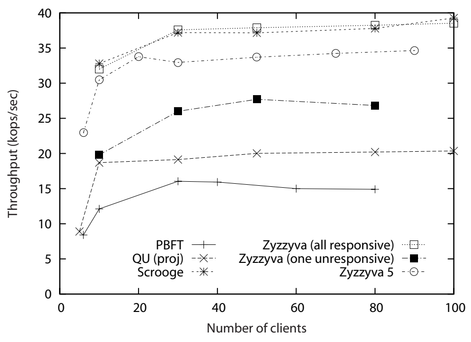

Figure 3.3: Throughput for 0/0 microbenchmark without batching and with $f = 1$.

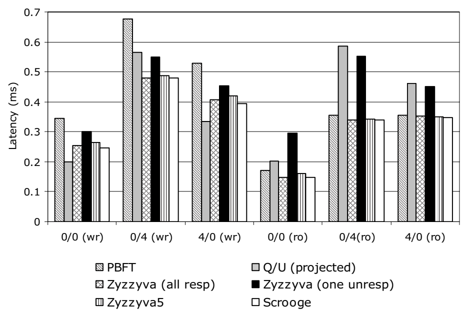

Figure 3.4: Latency for different benchmarks with a single client and no batching.

through a Gigabit switched star network. All servers are single-threaded processes. Nodes run Fedora Linux 8 with kernel version 2.6.23. We use MD5 to compute MACs and the AdHash library for incremental hashes as in [CL99; KAD$^+$07]. For performance stability, measurements are initiated after the execution of the first 10,000 operations, and are stopped after the successive 10,000 operations. We use the same X/Y micro-benchmark used by the authors of PBFT [CL99], where X and Y are the size (in KB) of client requests and replica replies respectively. We consider scenarios where all replicas are responsive and where one replica is initially crashed.

Throughput We first examine the throughput of Scrooge. Figure 3.3 shows the throughput achieved by the 0/0 micro-benchmark without batching. Scrooge is the protocol which achieves the highest throughput with the lowest, and in this case minimal, number of replicas. Zyzzyva5 displays similar trends but a slightly lower peak throughput. This is probably due to the use of a larger number of replicas, which forces the primary to calculate a higher number of MACs (40% more than Scrooge) to authenticate order request messages. Zyzzyva can perform as well as Scrooge only in runs with all responsive replicas because it cannot otherwise use speculation. In runs with one unresponsive replica, the peak throughput improvement of Scrooge over Zyzzyva is more than one third. PBFT has lower peak throughput because it calculates at least twice as many MACs as Scrooge and has quadratic message complexity.

If read-only requests with one unresponsive replica are considered, the difference becomes even more evident because Zyzzyva is not able to use the read optimization, as previously discussed. Even using batches of size 10, Zyzzyva achieves 52 kops/s peak throughput in presence of read-only workloads, whereas Scrooge achieves a peak of 85 kops/s.

Latency The latency of different protocols using different micro-benchmarks is shown in Figure 3.4. Scrooge performs in line with Zyzzyva5 with all micro-benchmarks. PBFT has approximately 40% higher latency than Scrooge for write requests and similar latency as Scrooge for read-only requests. Zyzzyva suffers a significant performance degradation in runs with unresponsive replicas. In case of write requests the difference with Scrooge ranges between 14% for the 0/4 case to 22% for the 0/0 case. The difference becomes much higher for read-only operations because unresponsive replicas disable the read-only optimization. The time a client needs to wait when it tries to use the read optimization without success depends on the timer settings of the client and is hard to evaluate. Figure 3.4 only considers for Zyzzyva the optimistic latency given by processing read requests upfront as normal writes. Even in this scenario, the latency of Zyzzyva compared to Scrooge is 29% higher in the 0/0 case and up to 98% higher for the 4/0 case.

Figure 3.5 illustrates how latency scales with the throughput when batching is not used. Scrooge is the protocol achieving the best latency at lowest, and in this case minimal, cost. Scrooge and Zyzzyva5 have almost equal measurement results. Zyzzyva displays higher latency ($\sim$ 0.9 kops/sec) in runs with unresponsive replicas and 10 clients.

Fault scalability A fault scalable replication protocol keeps costs low when the number of replicas, and thus of tolerated faults, grows [AEMGG$^+$05]. Scrooge is the most fault-scalable primary-based protocol in presence of unresponsive replicas. In Scrooge a primary computes $2 + (4f - 1)/s$ MACs operations per request if $b = f$ and s is the size of a batch. This is also the number of messages sent and received by the primary.

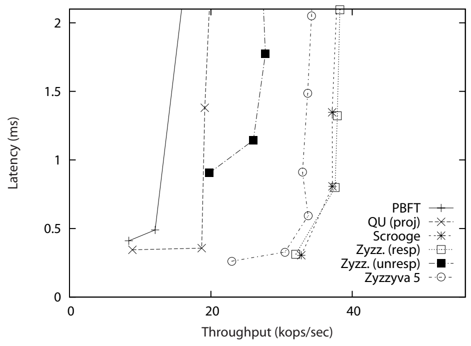

Figure 3.5: Latency-throughput curves for 0/0 microbenchmark without batching and with $f = 1$.

Zyzzyva has a slightly lower overhead in fault-free runs, $2 + 3f/s$. Scrooge is more scalable than PBFT $(2 + 8f/s)$, Zyzzyva5 $(2 + 5f/s)$ and Zyzzyva with one unresponsive replica $(2 + 5f/s)$. In Q/U the bottleneck replica makes only 2 MACs operations per request.

Scrooge uses $1 + 3f + (4f - 1)/s$ messages per request, similar to Zyzzyva in fault-free runs $(2 + 3f + 3f/s)$, Zyzzyva5 $(2 + 4f + 5f/s)$ and, with $s = 1$, Q/U $(2 + 8f)$. With unresponsive replicas, PBFT and Zyzzyva with commit optimization have quadratic complexity. Without commit optimization, Zyzzyva has lower message complexity but also significantly lower performance [KAD+07].

3.6 Chapter Summary

BFT state machine replication requires making a tradeoff between optimal performance and replication costs. Scrooge mitigates this tradeoff through two novel techniques: replier quorums and message histories. Compared with Scrooge, PBFT is less performant, Zyzzyva matches its performance only in fault-free runs, and Zyzzyva5 has similar performance but higher replication costs. In systems where tolerating any number of crashes but only one Byzantine failure is sufficient, Scrooge is the best choice as it is always fast and uses a minimal number of replicas.

Chapter 4

BFT with Trusted Components

Fault tolerant distributed protocols typically utilize a homogeneous fault model, either fail-crash or fail-Byzantine, where all processors are assumed to fail in the same manner. In practice, due to complexity and evolvability reasons, only a subset of the nodes can actually be designed to have a restricted, fail-crash failure mode, provided that they are free of design faults. Based on this consideration, this thesis proposes a *fail-heterogeneous architectural model* for distributed systems that considers two classes of nodes: (a) full-fledged *execution nodes*, which can be fail-Byzantine, and (b) lightweight, validated *coordination nodes*, which can only be fail-crash. This chapter also introduces HeterTrust, a practical trustworthy service replication protocol. It has a low latency overhead, requires few execution nodes with diversified design, prevents intruded servers from disclosing confidential data, and can withstand DoS attacks.

4.1 Introduction

This chapter describes HeterTrust, a practical trustworthy state machine replication protocol for asynchronous systems. HeterTrust only relies on Ω for progress. It uses dedicated coordination nodes (called coordinators in the following) to order client requests and filter replies coming from the execution nodes (called servers in the following) for confidentiality. Coordinators are more trustworthy than servers and can only fail by crashing. Figure 4.1 illustrates a typical fail-heterogeneous architecture. Three-tiered Web-scale systems running relatively unreliable applications software in the second and third tier and and more stable Web servers in the first tier such as those described in [ZBWM08] can be modeled as a fail-heterogeneous architecture. The description of the algorithm uses some Paxos-related terminology taken from [Lam01] and reviewed in section 2.1.2.

4.2 System Model

The system is composed of c fail-crash *coordinators*, s fail-Byzantine *servers* and a bounded number of authenticated *clients*. If the system is to tolerate up to g coordinator crashes and up to f Byzantine servers, it is assumed to have $c \geq 2g + 1$ coordinators and $s \geq 2f + 1$ servers. The protocol tolerates any number of malicious clients. Correct clients have only one pending operation at a time. We also assume that all client operations are uniquely identifiable. This can be easily implemented by having clients attach their unique id and a monotonically increasing local timestamp to each operation. Note that if trusted coordinators are sub-components of execution servers, as assumed in [CNV04], only a majority of correct replicas is required. HeterTrust would then achieve the same redundancy reduction as [CNV04] but without assuming synchronous channels.

Participants communicate through an asynchronous, unreliable network. Channels are authenticated, that is, if a correct process receives a message from a correct sender then the message was actually sent by the correct sender. Channels between any pair of correct hosts are fair-lossy, i.e., they eventually deliver messages that are repeatedly resent. Coordinators use an Ω failure detector to eventually elect a single leader among them. In order to provide confidentiality and protect from DoS attacks, coordinators must be physically interposed between clients and servers (see Figure 4.1).

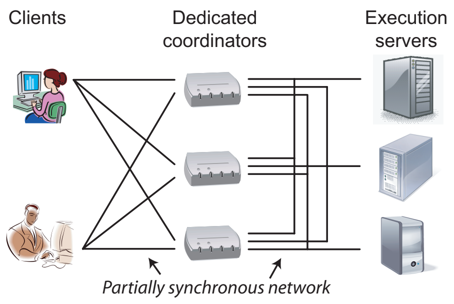

Figure 4.1: The fail-heterogeneous architecture used by HeterTrust

4.3 Service Properties

The protocol allows clients to send requests to the trustworthy replication service through the coordinators. The leader coordinator assigns a progressive sequence number to each received request and sends it to the servers, which execute the request and send it back, together with the reply, to all coordinators. These filter out spurious or incorrect replies and forward the correct ones to the client, which delivers it (see Figure 4.2). Formally, the properties provided by the trustworthy state machine replication service are the following (adapted from [MBTPV06]):

Termination: If a correct client *cl* invokes an operation *op* then it eventually delivers a reply *repl*.

Uniform Agreed Order: If a correct server commits an operation *op* as the k^{th} operation, then every correct server that commits the k^{th} operation must commit *op* as the k^{th} operation.

Update Integrity: For each operation *op*, every correct server commits *op* at most once, and only if a client has issued *op*.

Response Correctness: If a client receives an operation *op*, then the client has sent *op* and at least one correct server has sent *repl* as a reply to *op* in the commit order.

Termination is the liveness condition of the service. *Uniform Agreed Order* prevents correct servers from diverging and is sufficient for linearizability, i.e., clients sending concurrent requests to the service can observe the same course of action. *Update Integrity* guarantees that operations take effect exactly-once and that operations are not forged. *Response Correctness* enforces both integrity and confidentiality as it requires filtering out spurious and incorrect replies. These may be originated from Byzantine servers also to convey confidential information.

4.4 The HeterTrust Protocol

The algorithms executed by the clients, by the coordinators during normal operations and during recovery, and by servers are Algorithm 4, 5, 7 and 6 respectively. Table 4.1 explains the local variables used by the processes and their initial values. After an overview of the algorithm in section 4.4.1, sections 4.4.2 and 4.4.3 describe the normal operations of the protocol and the recovery from leader crashes, respectively.

4.4.1 Overview

Beyond showing the use of trusted coordinators, HeterTrust introduces two innovative algorithmic methods. First, it lets requests be executed before they are learned, and lets the client act as a learner. In this sense, HeterTrust is similar to speculative algorithms developed in parallel to this thesis' work, such as Zyzzyva [KAD$^+$07]. Second, HeterTrust includes algorithmic mechanisms to tolerate Denial of Service (DoS) attacks.

Using clients as learners HeterTrust lets clients act as learners in order to reduce the latency of the algorithm. Clients deliver replies to their operations before coordinators and servers know that they are committed. Servers use tentative executions, similar to PBFT. With tentative executions, servers can execute a single operation before knowing that it is committed. The Zyzzyva protocol pushes these ideas further by letting replicas execute multiple requests tentatively. The client in this case must learn that all replicas agree on all operation up to its request. This is achieved by using history digests. Different from HeterTrust, Zyzzyva considers a weaker, homogeneous Byzantine model where there are no trusted coordinators. On the other hand, it requires a higher number of replicas.

DoS tolerance HeterTrust is safe in periods of asynchrony but relies on additional synchrony for liveness [FL81]. If one can expect that these properties are generally met by the network in a benign (i.e., crash-only) environment, they cannot be in general preserved if an attacker is able to introduce "malicious asynchrony" and make communication unreliable by means of DoS attacks. Therefore, unless specific countermeasures are taken, replicated services can be made unavailable through DoS attacks. In a fail-heterogeneous architecture, crash-only coordinators can participate in the consensus protocol as filtering elements to accurately recognize and handle malicious traffic in an end-to-end manner on the whole protocol stack up to the state machine replication level. This overcomes the major limitation of network layer

Name	Description	Initial value
	coordinators	
k	next sequence number	0
$accval[k']$	accepted request	$\perp$
$learntval[k']$	learnt request	$\perp$
$maxProp$	maximum proposal number observed	0
$prop$	proposal number	i
$propval[k']$	proposed value	$\perp$
$Retr$	set of sequence numbers with retrievable requests	$\emptyset$
	execution servers	
$bComm[k']$	buffered request to be committed	$\perp$
$bProp[k']$	buffered request to be executed	$\perp$
$endGap$	end of a gap in the requests received by a server	-
$lastComm$	sequence number of the last committed request	0
$maxProp$	maximum proposal number observed	0

Table 4.1: Global Variables (for sequence number k')

DoS detection mechanism: the lack of information on viable communication patterns at the application layer.

In closed and controlled networks, such as LANs, *network layer* attacks can be prevented by eliminating network-level shared resources (e.g., network links and interfaces) and by establishing dedicated links between protocol participants, possibly including clients and coordinators. At a higher lever of the protocol stack, HeterTrust includes algorithmic mechanisms to prevent faulty execution servers from launching *replication-layer* DoS attacks. The core idea is that coordinators are able to determine when requests are completed. This is because, using the Paxos terminology, coordinators act as learners.

Execution servers are only connected to the coordinators and are not connected with each other. In normal runs, servers only reply to requests sent by the coordinators. The only case when servers may need to initiate communication with other entities is when they have temporarily lagged behind the other processes and have missed requests that were executed by other servers. In this case, servers may need to fetch requests to participate again in the algorithm.

In HeterTrust, servers that fetch requests only have to communicate with a single coordinator. This enables each single coordinator to use conventional rate-limiting techniques in isolation, without requiring expensive communication. Rate-limiting can thus be used to handle both faulty servers trying to fetch too many requests, and faulty clients trying to send too many requests to the system. For clients, rate-limiting can be used to limit both the number of different request they submit and the frequency of re-sending previous requests. Different from servers, single clients can be rate-limited without compromising progress for other clients. The description of the algorithm

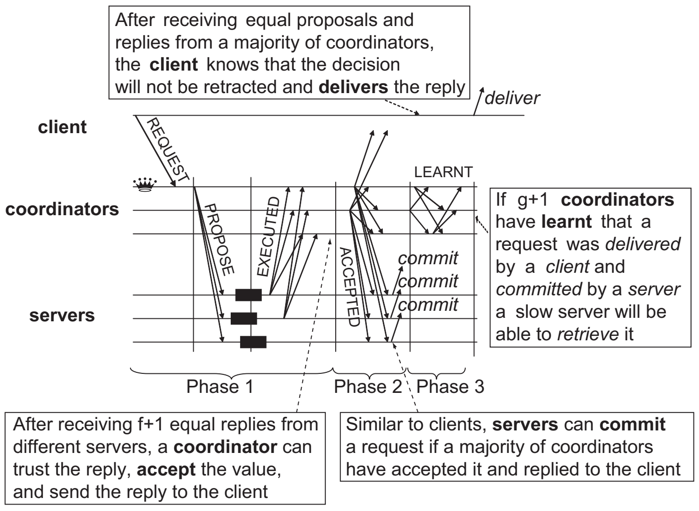
After receiving equal proposals and replies from a majority of coordinators, the **client** knows that the decision will not be retracted and **delivers** the reply
deliver
client
REQUEST
coordinators
servers
PROPOSE
EXECUTED
ACCEPTED
LEARNT
commit
commit
commit
If g+1 **coordinators** have **learnt** that a request was *delivered* by a *client* and *committed* by a *server* a slow server will be able to *retrieve* it
Phase 1
Phase 2
Phase 3
After receiving f+1 equal replies from different servers, a **coordinator** can trust the reply, **accept** the value, and send the reply to the client
Similar to clients, **servers** can **commit** a request if a majority of coordinators have accepted it and replied to the client

Figure 4.2: HeterTrust: Normal operations

abstracts away rate-limiting issues.

Work on DoS tolerance subsequent to the work of this thesis has developed DoS-tolerance mechanisms for homogeneous Byzantine models [ACKL08; CWA$^+$09].

4.4.2 Normal Operations

This section describes runs where there is no message loss and there is a single correct leader coordinator which has completed recovery and which is trusted by all correct coordinators. In these runs, the protocol proceeds through three phases upon the reception of a request from a client (see Figure 4.2). In Phase 1, it tries to provide a quick answer to the client. In Phase 2, it goes through an additional coordination step to let coordinators and servers know about the reply (potentially) delivered by the client. Finally, in Phase 3 it ensures that slow servers can directly retrieve old requests from at least one coordinator without triggering other instances of the agreement protocol.

Phase 1: Replying to clients. When the client wants the service to perform an operation *op*, it initiates the protocol by sending a REQUEST message to all coordinators (lines 4.1–4.3). Clients periodically resend pending requests to the coordinators (lines 4.11–4.13).

When the leader coordinator receives a request (line 5.1), it forms a proposal by attaching a proposal number *prop* to the request. The proposal number is used by the other coordinators to discard messages coming from old leaders. Each coordinator is assigned a partition of the set of positive

Algorithm 4: Client *cl*

```
4.1  upon invoke(op)
4.2      send (REQUEST, op) to all coordinators;
4.3      start timer;
4.4
4.5  upon receive (ACCEPTED, k, op, prop, repl) message from coordinator co
4.6      if op is pending then
4.7          if received (ACCEPTED, k, op, prop, repl) from $\lceil (c+1)/2 \rceil$ coordinators then
4.8              stop timer;
4.9              deliver(repl);
4.10
4.11 upon timeout
4.12     resend the pending REQUEST message to all coordinators;
4.13     start timer;
4.14
```

integers. Upon election, a leader coordinator increases its proposal number until it becomes the highest observed by a sufficient number of others participants, which will then *endorse* it (see section 4.4.3 for details). A leader proposes only a bounded number of requests in parallel and queues the remaining requests (at most one for each client, line 5.3).

The operation is then given an increasing sequence number k, stored in $propval[k]$, and sent in a PROPOSE message to all servers. Following the terminology of [Lam01] the request is now *proposed.* The sequence number will be used by each correct server to order the execution of requests and thus to keep a consistent state with the other correct servers. As long as there is only one leader coordinator, a single request will be assigned a unique increasing sequence number.

On receiving a PROPOSE message from the current leader, the servers produce a reply *repl* (lines 6.1–6.10). New requests are only *tentatively* executed, i.e., the changes to the service state are written in a temporary log before being *committed.* If the leader crashes the new leader can change the order of some requests, and this can cause tentative executions to *roll back.* Otherwise, tentative executions are eventually and definitively committed. HeterTrust admits only one non-committed tentative execution at a time. However, it is possible to extend the algorithm to execute multiple consecutive tentative request as done in [KAD$^+$07]. This would require coordinators to attach the digest of their current history onto ACCEPTED messages. Clients and servers can check these digests to make sure that the histories of all coordinators are consistent up to the considered operation.

Servers should only accept messages from the latest leader. For this purpose, they store the highest proposal number they have observed ($maxProp$). They also store the sequence number of the last committed request ($lastComm$) and only execute the next request (line 6.8). Requests

Algorithm 5: Coordinator i - normal operations

```
5.1  upon receive (REQUEST, $op$) message from client $cl$
5.2      if $\Omega = i$ then
5.3          if never sent a (PROPOSE, $k'$, $op$, $prop$) message for some value of $k'$ then
5.4              $k \leftarrow k + 1$;
5.5              $propval[k] \leftarrow op$;
5.6              send (PROPOSE, $k$, $op$, $prop$) to all servers;
5.7              start timer;
5.8          else
5.9              resend the prior (PROPOSE, $k'$, $op$, $prop$) message to all servers;
5.10             start timer;
5.11
5.12 upon receive (EXECUTED, $k'$, $op$, $prop$, $repl$) message from a server
5.13     if $prop \geq maxProp$ then
5.14         $maxProp \leftarrow prop$;
5.15         if received (EXECUTED, $k'$, $op$, $prop$, $repl$) messages from $f + 1$ servers then
5.16             $accval[k'] \leftarrow (op, prop)$;
5.17             send (ACCEPTED, $k'$,$op$,$prop$,$repl$) to client $req.cl$;
5.18             send (ACCEPTED, $k'$,$op$,$prop$) to all coordinators and servers;
5.19
5.20 upon received (ACCEPTED, $k'$, $op$, $prop$) message from a coordinator
5.21     if received (ACCEPTED, $k'$, $op$, $prop$) messages from $\lceil (c + 1)/2 \rceil$ coordinators then
5.22         $learntval[k'] \leftarrow (op, prop)$;
5.23         send (LEARNT, $k'$, $op$, $prop$) to all coordinators;
5.24
5.25 upon received (LEARNT, $k'$, $op$, $prop$) message from a coordinator
5.26     if ($learntval[k'] = \bot$) then
5.27         $learntval[k'] \leftarrow (k', op, prop)$;
5.28     if received (LEARNT, $k'$, $op$, $prop$) messages from $g + 1$ coordinators then
5.29         $Retr \leftarrow Retr \cup \{k'\}$;
5.30
5.31 upon received (RETRIEVE, $k'$) message from server $se$
5.32     if ($learntval[k'] \neq \bot$) then
5.33         send (LEARNT, $learntval[k']$) to $se$;
5.34
5.35 upon timeout
5.36     if $\Omega = i$ then
5.37         foreach $k' \notin Retr$: $propval[k'] \neq \bot$ do
5.38             send (PROPOSE, $k'$, $propval[k']$, $prop$) to all servers;
5.39         start timer;
5.40
```

with higher sequence number are buffered in $bProp$ unless they have been already buffered or if they come from a previous leader (lines 6.11–6.12).

A server first checks if the operation it receives has already been executed (line 6.4), and retrieve the previous reply in this case. Servers cache the last operation executed for each client. If a server receives a new operation with sequence number following the last committed operation, it obtains the reply by tentatively executing the operation possibly after performing a rollback of previous tentative executions (line 6.9). Servers attach the reply, together with the proposal, in an EXECUTED message sent to all the coordinators.

The coordinators ignore proposals from previous leaders (line 5.13). They also filter out malicious and spurious replies from servers by waiting for $f+1$

Algorithm 6: Server

```
6.1  upon receive (PROPOSE, $k'$, $op$, $prop$) message from a coordinator
6.2      if $prop \geq maxProp$ then
6.3          $maxProp \leftarrow prop$;
6.4          if $op$ has been already locally executed with sequence number $k'$ then
6.5              $repl \leftarrow$ output of the last local execution of $op$;
6.6              send (EXECUTED, $k'$, $op$, $prop$, $repl$) to all coordinators;
6.7          else
6.8              if $k' = lastComm + 1$ then
6.9                  $repl \leftarrow$ execute($op$);
6.10                 send (EXECUTED, $k'$, $op$, $prop$, $repl$) to all coordinators;
6.11             else if $k' > lastComm + 1 \wedge (bProp[k'] = \perp \vee prop > bProp[k'].prop)$ then
6.12                 $bProp[k'] \leftarrow (op, prop)$;
6.13
6.14 upon receive (ACCEPTED, $k'$, $op$, $prop$) message from a coordinator
6.15     if received (ACCEPTED, $k'$, $op$, $prop$) messages from $\lceil (c+1)/2 \rceil$ coordinators then
6.16         if $k' = (lastComm + 1)$ then
6.17             learnt($k'$, $op$, $prop$);
6.18         if $k' > (lastComm + 1)$ then
6.19             $bComm[k'] \leftarrow (op, prop)$;
6.20
6.21 procedure learnt($k'$, $op$, $prop$)
6.22     if $op$ has been already locally executed then
6.23         $repl \leftarrow$ output of the last local execution of $op$;
6.24     else
6.25         $repl \leftarrow$ execute($op$);
6.26     commit the execution of $op$;
6.27     $k'' \leftarrow k' + 1$;
6.28     $lastComm \leftarrow lastComm + 1$;
6.29     if $bComm[k''] \neq \perp$ then
6.30         learnt($k''$, $bComm[k''].op$, $bComm[k''].prop$);
6.31     else if $bProp[k''] \neq \perp$ then
6.32         trigger event handler for receipt of a (PROPOSE, $k''$, $bProp[k''].op$, $bProp[k''].prop$)
             message from a coordinator;
6.33
6.34 upon timeout
6.35     $endGap \leftarrow \min\{k' \mid (k' > lastComm) \wedge ((bProp[k'] \neq \perp) \vee (bComm[k'] \neq \perp))\}$ ;
6.36     foreach $k' \in [lastComm + 1, endGap - 1]$ do
6.37         send (RETRIEVE, $k'$) to all coordinators ;
6.38     start timer;
6.39
```

equal EXECUTED messages (line 5.15). This ensures that the reply was sent by at least one correct server and that it is an actual reply to a request proposed by the leader. In this case coordinators *accept* [Lam01] the proposal for a sequence number k' by storing it in the variable $accval[k']$ (line 5.16) . It then notifies, through an ACCEPTED message, all coordinators, servers and the client $req.cl$ which issued the request (lines 5.17–5.18). The ACCEPTED message sent to the client also contains the correct reply.

When the client receives an ACCEPTED message (line 4.5) for an ongoing request (line 4.6), it knows that the reply to its request was tentatively executed by at least one correct server. However, such a reply will only be delivered after it is guaranteed that this tentative execution will not roll back.

As discussed in section 4.4.3, the recovery protocol ensures that if a request is *chosen* for a sequence number [Lam01], i.e., it is contained in a proposal that is accepted by a majority of coordinators, then its execution will never be rolled back even if the leader and other coordinators crash. The client thus waits until it receives ACCEPTED messages for the same proposal from a majority of coordinators before delivering the reply (line 4.9). Thus, after four communication steps a client can deliver the reply.

If confidentiality is not required, one communication step can be saved by having servers send EXECUTED messages directly to the clients, which can thus filter out incorrect replies by waiting for $f + 1$ equal replies. In this case, the leader sends PROPOSE messages to servers and coordinators in the same communication step, and clients will deliver a correct reply only after receiving ACCEPT messages by a majority of coordinators.

Phase 2: Committing the reply. In order to ensure progress, coordinators take additional steps to guarantee that the servers can commit tentative executions. Coordinators and servers try to determine if a request was chosen for a sequence number and is therefore indissolubly bound to it. Similar to clients, they do this by waiting for ACCEPTED messages by a majority of coordinators (lines 5.20–6.15 and 6.14–6.17). When this happens, the request is *learnt* for a sequence number [Lam01]. Coordinators store learnt requests for sequence number k' in the variable $learntval[k']$ (line 5.22), and communicate this to all the other coordinators (line 5.23). Coordinators can also learn that a request was chosen by receiving a LEARNT message (line 5.25–5.27).

A server learns that a request was chosen (line 6.21) if it has sequence number $lastCommit + 1$. Commits for higher sequence numbers, as well as requests, are buffered in $bComm$ (line 6.19). If a chosen request has not already been executed, it is tentatively executed, after executing a roll back if necessary, and then committed (lines 6.22–6.26). Subsequently, further buffered requests for the next sequence numbers, which have been learnt or proposed, can be processed (lines 6.27–6.32).

Phase 3: Handling slow servers and message losses. Some servers might not learn that a request was chosen, either because they are slow or due to message losses. This prevents them from committing a tentative execution, and thus from executing further requests they receive. In this case the server sends a RETRIEVE message to the coordinators to learn the chosen request (lines 6.34–6.37 and 5.31–5.33). To guarantee that at least one coordinator will be able to reply to RETRIEVE messages, the leader has to keep sending PROPOSE messages and thus push protocol messages until it receives $g+1$ LEARN messages from different coordinators (lines 5.29 and (lines 5.35–5.38). One option for a server to recover an operation for a

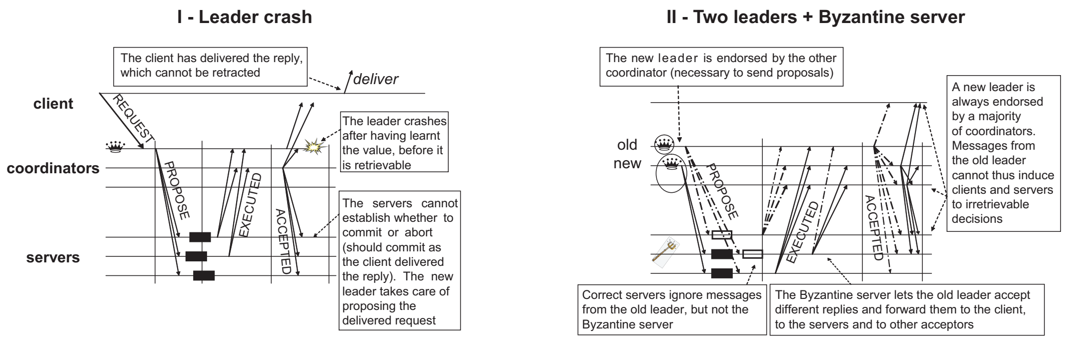
I - Leader crash
The client has delivered the reply, which cannot be retracted
deliver
client
REQUEST
coordinators
servers
PROPOSE
EXECUTED
ACCEPTED
The leader crashes after having learnt the value, before it is retrievable
The servers cannot establish whether to commit or abort (should commit as the client delivered the reply). The new leader takes care of proposing the delivered request
II - Two leaders + Byzantine server
The new leader is endorsed by the other coordinator (necessary to send proposals)
old
new
PROPOSE
EXECUTED
ACCEPTED
A new leader is always endorsed by a majority of coordinators. Messages from the old leader cannot thus induce clients and servers to irretrievable decisions
Correct servers ignore messages from the old leader, but not the Byzantine server
The Byzantine server lets the old leader accept different replies and forward them to the client, to the servers and to other acceptors

Figure 4.3: Two fail-prone scenarios

given sequence number would be to trigger a consensus instance. However, having coordinators act as learners prevents malicious servers from flooding the system by triggering consensus instances for request retrieval. A request is called *retrievable* if at least $g + 1$ coordinators have learnt it. Retrievable requests can be obtained by servers by contacting one correct coordinator, without initiating new consensus instances.

4.4.3 Recovery

Due to system asynchrony and crashes, the leader election protocol can output multiple coordinators as leaders, possibly at the same time. It is then necessary to prevent newly elected leaders from retracting decisions which already caused irreversible evolutions of the system state, such as the delivery of a reply to a client or the commit of an execution done by a server. In particular, if a request is chosen, i.e., it is accepted by a majority of coordinators, it is necessary to prevent new leaders from proposing different requests and having them accepted. Consider for example scenario (I) of Figure 4.3. In this case, the new leader must ensure that servers will commit the (chosen) request used to compute the reply delivered by the client. To guarantee this, the protocol adopts a recovery procedure (Algorithm 7) which is similar to the one used by the Paxos protocol [Lam98]. The similarity is natural given that only fail-crash participants (i.e. the coordinators) are directly involved.

Upon being elected, a leader selects a new proposal number and sends a QUERY message asking all other coordinators to endorse it (lines 7.1–7.10). It also asks them if (a) they have accepted some request $accval[k']$ for the sequence numbers k' that are not yet bound to a retrievable request in its local view (i.e., they are not in $Retr$), or (b) they know that there is a retrievable request for these numbers. Unless the other coordinators have already endorsed another leader with a higher sequence number (line 7.25),

```
Algorithm 7: Coordinator - recovery
7.1  upon $\Omega$ changes its value to $i$
7.2      query();
7.3
7.4  procedure query()
7.5      $prop \leftarrow$ minimum $p > maxProp_{co}$ of any correct coordinator $co$ such that $p \bmod i = 0$;
7.6      send (QUERY, $prop$, $Retr$) to all coordinators;
7.7      start timer-rec;
7.8
7.9  upon received (ENDORSE, $prop$, $Acc_{co}$, $Retr_{co}$) message from coordinator $co$
7.10     if $\Omega = i$ and received (ENDORSE, $prop$, $Acc_{co'}$, $Retr_{co'}$) from $\lceil (c+1)/2 \rceil$ coordinators $co'$
         then
7.11         stop timer-rec;
7.12         $Retr \leftarrow Retr \cup_{co} Retr_{co}$;
7.13         $maxRetr \leftarrow \max\{k' \mid (k' \in Retr) \wedge (\forall k'' \leq k', j \in Retr)\}$;
7.14         $maxAcc \leftarrow \max\{k' \mid \exists av, co'' : (k', av) \in Acc_{co''}\}$;
7.15         $propval \leftarrow \bot$;
7.16         foreach $k' \in [max(Retr) + 1, maxAcc] : k' \notin Retr$ do
7.17             $Proposals \leftarrow \{av \mid \exists co'' : (k', av) \in Acc_{co''}\}$;
7.18             if $Proposals \neq \emptyset$ then
7.19                 $propval[k'] \leftarrow av.op$ such that $av \in Proposals$ and has maximum $av.prop$;
7.20             else
7.21                 $propval[k'] \leftarrow$ no_op;
7.22         $k \leftarrow maxAcc + 1$;
7.23
7.24 upon receive (QUERY, $p'$, $Retr_l$) message from coordinator $co$
7.25     if $p' > maxProp$ then
7.26         $maxProp \leftarrow p'$;
7.27         $Retr \leftarrow Retr \cup Retr_{co}$;
7.28         $Acc \leftarrow \emptyset$;
7.29         foreach $k' : (k' \notin Retr) \wedge (accval[k'] \neq \bot)$ do
7.30             $Acc \leftarrow Acc \cup \{(k', accval[k'],)\}$;
7.31         send (ENDORSE, $endorse$, $Acc$, $Retr$) to coordinator $co$;
7.32
7.33 upon timeout-rec
7.34     if $\Omega = i$ then
7.35         query();
7.36
```

they endorse the new leader (line 7.26) and form their set of accepted requests *Acc* and retrievable requests *Retr* (which do not require further operations). They then send both sets to the new leader (lines 7.27–7.31).

Upon receiving ENDORSE messages from a majority of coordinators, the leader can start proposing requests (lines 7.12–7.21). For sequence numbers with an associated retrievable request, no operation is needed. For other sequence numbers where some accepted request is reported, the new leader must send its proposals without contradicting previously chosen requests. This is done by selecting the proposal from the latest previous leader, i.e., the one with the highest proposal number (lines 7.19). Gaps are filled with special *no_op* requests (line 7.21). If a request proposed from a certain leader is accepted by a majority of coordinators, each subsequent leader will receive notification of it in at least one ENDORSE message and select it for proposal.

This guarantees that chosen requests are not overwritten.

For progress, the leader has to send a proposal number that is higher than any proposal number observed by any correct process. This can be easily guaranteed by having each coordinator *co* send a message to a leader l when they receive a message from l with a proposal number lower than $maxProp_{co}$. The leader in this case starts recovery again with a proposal number higher than $maxProp_{co}$. For simplicity, we abstract away these details in the pseudocode and assume that the proposal number selected by the leader is high enough (line 7.5).

It is surprising how efficiently a Paxos-like recovery protocol under the fail-heterogeneous model tolerates the effect of Byzantine faults at the servers. For example, in scenario (II) of Figure 4.3, two leaders are simultaneously present and a Byzantine server sends them inconsistent information. Each leader waits for an endorsement from a majority of coordinators before issuing proposals, and Byzantine servers are not involved in this decision. Although servers can forward messages from an old leader to a minority of coordinators and have them accepted, these coordinators cannot induce correct clients and servers to take wrong delivery or commit actions.

4.4.4 Garbage Collection

Servers can discard all data regarding sequence numbers of committed requests. Coordinators could as well garbage-collect the data structures of sequence numbers of retrievable request, but they need to indefinitely keep them in $learntval[i]$ to reply to RETRIEVE messages from slow servers. To avoid this, a simple checkpointing protocol is used. This is not included in the previous algorithms, but it is described briefly in the following.

When the commit procedure at an execution server commits a request with sequence number i such that $(i \bmod k) = 0$ for a given checkpoint frequency k, it produces a tentative checkpoint of the local service state, calculates a digest of it and sends a (CHECKPOINT, i, *cpd*) message with the MAC of the digest *cpd* to all coordinators. After receiving $f + 1$ such equal messages from different servers, coordinators know that at least one correct server has an available checkpoint of the service state up to sequence number i. They then send an (ACKCP, i) message to all servers. When a server receives $g + 1$ such messages, it can delete the previous checkpoints, complete the commit procedure, and start processing the next executable requests.

A coordinator receiving a checkpoint digest from $f+1$ servers for sequence number i knows that it can garbage collect entries previous to $learntval[i]$ as, if necessary, slow servers trying to retrieve old chosen requests can be sent a complete checkpoint. However, to minimize state transfers, it tries to reply with simple requests when possible. Therefore, it only deletes entries of learnt requests in the array *learntval* for sequence numbers preceding the prior checkpoint, i.e., prior to $i-k+1$. A slow server receives the checkpoint state for sequence number i only if it tries to retrieve a request for a sequence number $j \leq i-k$. In this case, coordinators obtain the checkpoint state by sending a (QUERYCP, i) message to all servers until they receive at least one correct checkpoint, which is recognized using the digest. They then store the checkpoint state (at most one at a time) for further requests, and send it to the slow server.

4.5 Chapter Summary

This chapter introduced a new fail-heterogeneous architectural model, which represents an intermediate step between benign fail-crash models and conservative fail-Byzantine models. It is based on a separation of concerns between unconstrained execution nodes and lightweight coordination nodes, with reduced functionalities and thus restricted failure mode. The chapter showed how new Byzantine-fault tolerant replication algorithms under the new model can be developed that keep many advantages of fail-crash protocols, while tolerating more severe failures at the server nodes providing the service of interest. It introduces the HeterTrust protocol, which allows an efficient communication pattern similar to a crash-only protocol, but still ensures properties, such as confidentiality, that are extremely expensive to provide in a homogeneous fail-Byzantine model. Last but not least, HeterTrust reduces the number of required replicas with diversified design.

HeterTrust introduces some interesting algorithmic ideas. It lets clients act as learners to reduce the latency of the algorithms, leveraging the technique of tentative executions introduced in [CL99] and later developed in [KAD$^+$07]. It introduced the problem of tolerance to DoS attacks, a problem developed in subsequent work [ACKL08; CWA$^+$09]. The use of trusted subcomponents in untrusted nodes in asynchronous systems, has been discussed by work developed in parallel [CMSK07] or after [LDLM09] the work reported in this thesis.

Chapter 5

Eventual Linearizability

Linearizability is the strongest known consistency property of shared objects. In asynchronous message passing systems, Linearizability can be achieved with $\Diamond\mathcal{S}$ and a majority of correct processes. This thesis introduces the notion of *Eventual Linearizability*, the strongest known consistency property that can be attained with $\Diamond\mathcal{S}$ and *any number* of crashes. This chapter shows that linearizable shared object implementations can be augmented to support *weak* operations, which need to be linearized only eventually. Unlike *strong* operations that require to be always linearized, weak operations terminate in worst case runs. However, there is a tradeoff between ensuring termination of weak and strong operations when processes have only access to $\Diamond\mathcal{S}$. If weak operations terminate in the worst case, then this chapter shows that strong operations terminate only in the absence of concurrent weak operations. Finally, this chapter shows that an implementation based on $\Diamond\mathcal{P}$ exists that guarantees termination of *all* operations.

5.1 Definitions

This section first defines a model of concurrent executions. Next, it defines Eventual Linearizability and show that, like Linearizability, it is local and nonblocking.

5.1.1 Model of Concurrent Executions

This chapter considers concurrent systems consisting of a set of processes $\{p_i \mid i \in [0, n-1]\}$ accessing a set of shared objects. Processes interact with objects through *operations*. An execution is a history consisting of a finite sequence of operation *invocation* and *response events* taking place at a process and referring to an object. Invocations contain the *arguments* of the operation, while responses contain the *results* of the operation. All operations are unique and are ordered in the history according to the time of their occurrence. The presence of a global clock providing a time reference for the whole system is assumed, which starts from 0 and is often referred to as *real-time order*. Processes do not have access to this clock. Given a history H and a process p_j (resp. an object x), $H|j$ (resp. $H|x$) denotes the restriction of H to call and response events of p_j (resp. on x).

A history is *sequential* if (i) the first event is an invocation, (ii) all invocation events, except possibly the last, are immediately followed by the response event for the same operation, and (iii) response events are immediately preceded by the invocation event for the same operation. A sequential history H is *legal* if, for each object x, $H|x$ is correct according to the sequential specification of x. The relation $<_H$ denotes the order of operations defined by a sequential history H. A *sequential permutation* of a history H is a sequential history obtained by permuting the events of H. A history that is not sequential is called *concurrent*. An operation is called *completed* if the history includes an invocation and a completion event for it. For a history H, *completed*(H) denotes the subsequence of events in H related to all completed operations. A history is *well-formed* if the subhistory of events of each process is sequential. All histories are assumed to be well-formed.

5.1.2 Definition of Eventual Linearizability

Eventual linearizable implementations need to always ensure some minimal weak consistency property that rules out arbitrary behaviors. For each history H, it requires that the response to every completed operation o of every process p_i is the result of a legal sequential history $\tau(i, o)$. The history $\tau(i, o)$

must terminate with o, it must consist only of operations invoked in H before o is completed, and it must include all operations observed by p_i before o.

Formally, a history H is *weakly consistent* if, for every process p_i and operation o completed by p_i in H, there exists a legal sequential history $\tau(i, o)$ such that: (i) the last event in $\tau(i, o)$ is a response event of o having the same result as the response event of o in H, (ii) every operation invoked in $\tau(i, o)$ is also invoked in H before o is completed, and (iii) for each operation o' invoked by p_i before o, $\tau(i, o') \subseteq \tau(i, o)$.[1]

This definition of weak consistency is very generic. It allows processes to ignore operations of other processes. Furthermore, subsequent serializations observed by a process can reorder previously-observed operations. Eventual Linearizability can be combined with stronger weak consistency semantic than this. For example, section 5.3.2 shows that it is possible to combine Eventual Linearizability with causal consistency [Lam78].

Eventual Linearizability requires all operations that are invoked after a certain time t to be ordered with respect to all other operations according to their real-time order. Pairs of operations invoked before t can be ordered arbitrarily. This requirement on the order is formalized by the following relation. Let H be a history and t a value of the clock. The irreflexive partial order $<_{H,t}$ is defined as follows: $o_1 <_{H,t} o_2$ iff o_2 is invoked after t and the response event of o_1 precedes the invocation event of o_2.

A *t-permutation* P of a history H is a legal sequential history that orders operations of H according to $<_{H,t}$. The results of operations in P do not have to match with those of the corresponding operations in H. Formally, the following two properties must hold for a legal sequential history P to be a t-permutation of H: (P1) an operation o is invoked in P if and only if o is invoked in H; (P2) $<_{H,t} \subseteq <_P$. It is worth noting that every well-formed history H has a t-permutation P for each value of t. However, not every well-formed history has a *linearization* as defined in [HW90].

Eventual Linearizability is a property of histories that may initially be weakly consistent but that eventually start behaving like in a linearization. This constraint is formalized as follows. A *t-linearization* L of a history H is defined as a t-permutation where the results of all operations invoked after t are the same as in H. Operations invoked before t may have observed inconsistent histories that do not correspond to any single legal sequential history. A history H is *t-linearizable* if there exists a t-linearization of H. Note that all well-formed histories having a linearization also have a t-linearization.

It is now possible to define Eventual Linearizability as follows.

[1]The $\subseteq$ notation is abused here to indicate that the set of operations of $\tau(i, o')$ is included in the set of operations of $\tau(i, o)$.

Eventual Linearizability: *An implementation of a shared object is eventually linearizable if all its histories are weakly consistent and t-linearizable for some finite and unknown time t.*

Linearizability differs from Eventual Linearizability because the convergence time t is known and equal to zero. In general, any form of t-linearizability where t is known can be easily reduced to Linearizability in systems where processors have access to a local clock with bounded drift. This is why the properties consider more general scenarios where t exists but is unknown. It is worth noting that, different from t-linearizability, Eventual Linearizability is a property of implementations, not of histories. In fact, all finite histories are trivially t-linearizable for some value of t larger than the time of their last event. Showing Eventual Linearizability on an implementation entails identifying a single value of t for all histories.

Eventual Linearizability has two fundamental properties of Linearizability. *Locality* implies that any composition of eventually linearizable object implementations is eventually linearizable. *Nonblocking* requires that there exist no history such that every extension of the history violates Eventual Linearizability.

Theorem 1. *Eventual Linearizability is nonblocking and satisfies locality.*

5.2 Implementations

Eventual Linearizability only requires that operations are linearized eventually. It can thus be implemented using primitives that are weaker than Consensus. This section identifies which properties must be satisfied by these primitives. It focuses on weak operations where Eventual Linearizability is sufficient. Strong operations are introduced in section 5.3. Many weakly consistent implementations provide properties such as *Eventual Serializability* [FGL$^+$96] or *Eventual Consistency* [SS05; Vog09]. This section shows that these properties are not sufficient to implement Eventual Linearizability, and therefore defines a stronger problem, called *Eventual Consensus*, that is stronger than Eventual Consistency but weaker than Consensus. It finally shows that Eventual Consensus is necessary and sufficient to implement Eventual Linearizability.

5.2.1 System Model for Implementations

This section considers shared object implementations using an underlying *consistency layer* to keep replicas consistent. If Linearizability is required for all operations then the consistency layer implements Consensus. The specifications defined in this section refer to properties of consistency layers, unlike Eventual Linearizability which is a property of shared object implementations. For simplicity, the discussion refers to implementations of a single shared object.

The interface of the consistency layer has two types of events: *submit events*, which are input events, and *delivery events*, which are output events. Submit events include as input value an operation on the shared objects; delivery events return a sequence of operations on the shared object. $S(i,t)$ denotes the last sequence delivered to process p_i at time $t > 0$ and define $S(i,0)$ to be equal to the empty sequence for each i. The processes interacting with the shared object can fail by crashing. If p_i is crashed at time t, $S(i,t)$ is the last sequence delivered by p_i before crashing. A submitted operation *terminates* when it is included in a sequence that is delivered at each correct process.

The consistency layer itself is implemented on top of an asynchronous message passing system with reliable channels. Implementations can use *failure detectors* [CT96; CHT96]. A failure detector $\mathcal{D}$ is a module running at each process that outputs at any time a set of process indices [CT96]. The classes of failure detectors used in this paper are defined in [CT96] and reviewed in section 2.1.1.

5.2.2 Eventual Consistency and Eventual Consensus

The formalization of Eventual Consistency given in the following builds upon the properties of Eventual Serializability [FGL+96] and Eventual Consistency [SS05] and is expressed in terms of a weakened form of Consensus. Like Eventual Serializability, it allows processes to temporarily diverge from each other on the order of operations and to eventually converge to a total order. Eventual Serializability supports defining precedence relations with each operation to constraint their execution order. These relations are typically used to specify causal consistency [FGL+96; LLSG92]. Since the discussion here focuses on Eventual Consistency properties, these orthogonal aspects are abstracted away.

Eventual Consistency: *A consistency layer satisfies Eventual Consistency if the following properties hold.*

Nontriviality: *For any process p_i and time t, every operation in $S(i,t)$ has been invoked at a time $t' \leq t$ and appears only once in $S(i,t)$;*

Set stability: *For any process p_i, if $t \leq t'$ then each operation in $S(i,t)$ is included in $S(i,t')$;*

Algorithm 8: An eventually linearizable implementation of a generic object using Eventual Consensus.

```
8.1  execute(o, H): returns the result of executing the sequence $H$ up to and including the operation $o$;
8.2  upon invoke ($o$)
8.3      $curr \leftarrow o$;
8.4      submit($o$);
8.5
8.6  upon deliver($H$)
8.7      if $curr \neq \perp \wedge curr \in H$ then
8.8          $r \leftarrow$ execute($curr$, $H$);
8.9          $curr \leftarrow \perp$;
8.10         complete ($o$,$r$);
8.11
```

Algorithm 9: Solving Eventual Consensus using an eventually linearizable implementation of an append/read sequence object.

```
9.1  append(o): appends an operation $o$ at the end of the sequence;
9.2  read(): returns the current value of the sequence;
9.3  upon submit ($o$)
9.4      append($o$);
9.5
9.6  upon periodic tick
9.7      $H \leftarrow$ read();
9.8      deliver ($H$);
9.9
```

Prefix consistency: *For any time t there exists a sequence of operations P_t such that:*
(C1) *For any correct process p_i, P_t is a prefix of $S(i,t')$ if $t \leq t'$;*
(C2) *P_t is a prefix of $P_{t'}$ if $t \leq t'$;*
(C3) *Every operation o submitted at time t' by a correct process is included in $P_{t''}$ for some $t'' \geq t'$.*

Note that property (C3) of prefix consistency implies *Liveness*, i.e., for any correct processes p_i and p_j and time t, every operation submitted by p_i at time t is included in $S(j, t_j)$ for some $t_j \geq t$.

This definition of Eventual Consistency is a relaxation of Consensus on sequences of operations [Lam05].[2] Consensus requires the same nontriviality and liveness properties as Eventual Consistency, but requires stronger stability and consistency properties. *Stability* requires that for any process p_i, $S(i,t)$ is a prefix of $S(i,t')$ if $t < t'$. *Consistency* requires that for any processes p_i and p_j and time t, one of $S(i,t)$ and $S(j,t)$ is a prefix of the other.

[2]The definition considers the case where all processes are proposers and learners. It also modifies nontriviality to rule out sequences with duplicates.

Set stability allows reordering the sequence of operations returned as an output, provided that all operations returned previously are included in the new sequence. Prefix consistency allows replicas to temporarily diverge in a suffix of operations. However, it requires eventual convergence among all replicas on a common prefix P_t of operations. Property (C1) of prefix consistency says that a common prefix P_t of operations has been delivered by each replica; (C2) constraints this prefix to be monotonically increasing; (C3) ensures that all completed operations are eventually included in the common prefix.

Eventual Consistency is not sufficient to implement Eventual Linearizability, not even for simple read/write registers, as shown in Theorem 2. This and the following results in this section consider a non-uniform notion of Eventual Linearizability, where operations invoked by faulty processes may never appear in the final t-linearization. The focus on non-uniformity is motivated by two observations. The first is that this strengthens the impossibility results of this chapter, while extending the possibility results to the uniform case is not difficult. The second observation is that, as it can be derived by using a simple partitioning argument, ensuring a uniform notion of Eventual Linearizability would require the existence of $f + 1$ correct processes to complete weak operations if f replicas can crash. The availability of $f + 1$ correct replicas for completing weak operations is not assumed by most replication algorithms implementing Eventual Consistency [SS05]. For example, the specification of Eventual Serializability [FGL$^+$96], which models the behavior of Lazy replication [LLSG92], does not distinguish between operations of correct and faulty processes. However, Lazy replication implements a non-uniform form of Eventual Serializability, where operations observed only by faulty replicas may never appear in the eventual serialization.

Some eventually consistent (or eventually serializable) algorithms ensure that *all* completed operations appear in the eventual serialization. The Zeno algorithm, for example, requires clients to contact a quorum of replicas in order to complete weak operations [SFK$^+$09]. This is needed to prevent clients from returning replies from Byzantine replicas. Dynamo implements uniformity by writing values to “sloppy quorums” that might not intersect with read quorums [DHJ$^+$07]. If f failures are to be tolerated, both these algorithms require that that at least one quorum of $f + 1$ correct replicas is always available even in worst case runs.

Theorem 2. *An eventually linearizable implementation of a single-writer, single-reader binary register cannot be simulated using only an eventually consistent consistency layer.*

The intuition for this result can be given by a simple example. Consider two processes p_0 and p_1 that share one single-writer, single-reader binary register holding a current value 1 at a given time t. Assume that p_0 is the writer of the register and p_1 is the reader. Process p_0 invokes a $write_0(0)$ operation after t. After this operation is completed, process p_1 invokes a $read_1()$ operation. Prefix consistency allows the consistency layer to delay convergence to a common prefix P_t for an arbitrarily long time. Before completing $read_1()$, p_1 may thus not distinguish this run from a run where $write_0(0)$ was never invoked. Therefore, $read_1()$ returns the previous value 1. A consistent ordering P_t of these two operations can be delivered by the consistency layer of both processes *after* both operations are completed. This is sufficient to satisfy Eventual Consistency. Such a pattern can occur after any finite time, making t-linearizability impossible for any t.

The key to achieve Eventual Linearizability is in strengthening stability. Assume in the previous example that the consistency layer is not allowed to change the order of the operations it has delivered after t. p_0 can complete its operation only after the consistency layer delivers a sequence containing $write_0(0)$. In order to prevent the consistency layer of p_0 from reordering its delivered sequence, the first non-empty consistent prefix $P_{t'}$ must include $write_0(0)$. This implies that the consistency layer of p_1 has to deliver $write_0(0)$ before $read_1()$ in order to preserve stability. p_1 can thus execute this sequence and return 0, respecting linearizability. In other words, an Eventually Consistent consistency layer satisfying eventual stability must eventually start to deliver all operations in a total order *before* the operations are completed. This total order also includes all the operations that have been submitted before t.

The previous example gives us the insight for the definition of Eventual Consensus. Different from Eventual Consistency, the delivered sequences eventually stop reordering operations that were previously delivered.

Eventual Consensus: *A consistency layer satisfies Eventual Consensus if Eventual Consistency and the following additional property hold:*

Eventual Stability: *There exists a time t such that for any times t' and t'' with $t \leq t' \leq t''$ and for any process p_i, $S(i, t')$ is a prefix of $S(i, t'')$.*

Implementing Eventual Consensus is both necessary and sufficient to achieve Eventual Linearizability for generic objects as shown in Theorem 3. This result reduces the problem of obtaining eventually linearizable shared object implementations to the problem of implementing a consistency layer satisfying Eventual Consensus. The following Theorem 3 shows the equivalence to Eventual Linearizability

Theorem 3. *Eventual Consensus is a necessary and sufficient property of a consistency layer to implement arbitrary shared objects respecting Eventual Linearizability.*

Algorithm 8 shows the sufficiency part of the result. Whenever an operation is invoked, it is submitted to the consistency layer. The operation is then completed as soon as a sequence containing the operation is delivered. The returned sequence is executed and the result is returned in a completion event. Before stability eventually holds, nontriviality and set stability are sufficient to satisfy weak consistency. As discussed in the previous register example, eventual stability ensures that processes eventually start delivering operations in the same total order, which is identified by the consistent prefix P_t, *before* the operations are completed. This allows implementing Eventual Linearizability.

Necessity is shown by Algorithm 9, which uses a shared sequence having an append and a read operation. Whenever an operation is submitted, it is appended onto the sequence. The object is periodically read and its value is delivered. The weak consistency property of the sequence is sufficient to ensure nontriviality and set stability. When the object starts to be eventually linearizable, all reads and appends are totally ordered in a legal sequential history. This ensures that eventually all operations are included in the same total order, as required by prefix consistency, and that read sequences that are delivered are never reordered in the future, as required by eventual stability.

5.3 Combination with Linearizability

The previous discussion has distinguished between strong operations that need to be linearized and weak operations that require to be eventually linearized. Strong operations are delivered only if Consensus is reached on the prefix including them as last operation. This is called a *strong prefix.* The specification of Eventual Consensus is extended accordingly.

Strong prefix stability: *For any process p_i, time t, strong operation s and sequence π, if $\pi\, s$ is a prefix of $S(i,t)$ and $t' \geq t$ then $\pi\, s$ is a prefix of $S(i,t')$.*

Strong prefix consistency: *For any processes p_i and p_j, time t, strong operations s_i and s_j and prefixes π_i and π_j, if $\pi_i\, s_i$ is a prefix of $S(i,t)$ and $\pi_j\, s_j$ is a prefix of $S(j,t)$ then one of $\pi_i\, s_i$ and $\pi_j\, s_j$ is prefix of the other.*

If all operations are strong, Eventual Consensus is equivalent to Consensus. One would desire to achieve termination of weak operations in all runs together with termination of strong operations in runs where Linearizability can be achieved. This section discusses impossibility and possibility results on this topic.

5.3.1 Impossibility Result

This section shows that even if a $\Diamond\mathcal{S}$ failure detector is given for termination of weak operations, strong operations cannot terminate in runs where consensus can be solved (see Theorem 4).

The intuition behind the impossibility lays in the concurrency between weak and strong operations. The impossibility proof constructs an infinite run where some strong operation s is never completed. For this, it considers an Eventual Consensus layer ensuring stability after a time t in a run where all events occur after the time t. Assume that a strong operation s is submitted by a correct process and that the processes are trying to reach consensus on a strong prefix $\pi\, s$. Let a submit event for an operation $w \notin \pi$ occur at a correct process p_i before consensus on $\pi\, s$ is reached. Process p_i cannot know whether consensus will terminate or not, as it accesses only failure detector $\Diamond\mathcal{S}$, but it must deliver weak operations in either case. Therefore, p_i cannot wait until consensus on $\pi\, s$ is reached before delivering w. p_i is thus forced to deliver w before consensus on $\pi\, s$ is reached. When consensus on $\pi\, s$ is reached, eventual stability forbids p_i to deliver $\pi\, s$ because w is not in π. Therefore, consensus needs to be reached on a new strong prefix $\varphi\, s$ with $w \in \varphi$. However, a new weak operation w' may be submitted before consensus on $\varphi\, s$ is reached. This pattern can be repeated forever. As a result, the strong operation s is never completed even if consensus can be solved.

This result highlights an implicit tradeoff in implementing Eventual Linearizability. As a consequence of the impossibility result, shared object implementations using $\Diamond\mathcal{S}$ can ensure Eventual Linearizability and give up termination of strong operations in presence of concurrent weak operations. Alternatively, they can choose to violate Eventual Linearizability in order to ensure termination of both weak and strong operations. In the latter case, it follows from the impossibility that Eventual Linearizability can be violated whenever there are concurrent weak and strong operations.

The proof of the following theorem describes asynchronous computations in terms of events as in [AW04]. *Input* events submitting operation o at p_i are denoted as $\mathit{submit}_i(o)$. An *output* event occurs when a sequence π is delivered. An operation is *delivered* when a sequence containing it is delivered. *Message receipt* events occur when a process receives a message. The occurrence of these events at a process p_i might *enable* the occurrence of *computation* events at p_i, which might in turn result in p_i sending new messages.[3] A message m is *causally dependent on an event* e if the computation event that generated m is causally dependent on e according to the classical definition of Lamport [Lam78].

Theorem 4. *In a system with $n \geq 3$ processes out of which f can crash, it is impossible to implement a consistency layer that satisfies the following properties using a failure detector $\Diamond\mathcal{S}$: (P1) termination of weak operations; (P2) termination of strong operations if $f < n/2$; and (P3) Eventual Consensus.*

Proof. Assume by contradiction that a consistency layer satisfying properties (P1), (P2) and (P3) exists. Let processes be partitioned into two sets, Π_m of size $\lfloor (n-1)/2 \rfloor$ and Π_M of size $\lceil (n+1)/2 \rceil$. By (P3), there exists a time t after which eventual stability holds for each run. Consider all runs where no process fails and where the $\Diamond\mathcal{S}$ modules of all processes suspect Π_M. This proof builds one such run σ that begins with an event $submit_h(s)$, with $p_h \in \Pi_M$ occurring after time t, where s is a strong operation. σ is an infinite and fair run that is built using an infinite number of finite runs σ_k with $k \geq 0$ in which s is never delivered by any process, thus violating (P2). Each run σ_k with $k > 0$ is built by extending σ_{k-1}. The run σ is the result of an infinite number of such extensions. Run σ is fair by construction because all messages sent in σ_{k-1} are received in σ_k, and because all enabled computation events occur.

Let M_k be the set of messages that are sent, but not yet received, in σ_k. For each σ_k, this proof shows by induction on k the following invariant (I): No process delivers s in σ_k or in any extension of σ_k where (i) all processes in Π_M crash immediately after σ_k, and (ii) all messages in M_k sent by processes in Π_M are lost.

First the case $k = 0$ is considered, and σ_0 is defined as follows. Let $submit_h(s)$ be the first and only input event of the system. Assume that no process crashes in σ_0. Assume also that no message is received in σ_0 and that all enabled computation events occur. Let M_0 be set of initial messages sent in σ_0.

It is easy to see that (I) is satisfied in σ_0. Since only a strong operation has been submitted, delivering s entails solving consensus on s by definition. Property (I) directly follows from the facts that no message is received in σ_0 and that consensus cannot be solved using $\Diamond\mathcal{S}$ in any extension satisfying conditions (i) and (ii) since $f \geq \lceil n/2 \rceil$ (see proof in [CT96]).

For the inductive step, σ_k is constructed for $k > 0$ by extending σ_{k-1}. Assume that no process crashes in σ_k and that $\Diamond\mathcal{S}$ permanently suspects

[3]If a process sends a message to itself, then the receipt of this message is considered as a local computation event.

Π_M. Let an event $submit_i(w_k)$ occur at a process $p_i \in \Pi_m$ after σ_{k-1}, where w_k is a weak operation that has never been submitted earlier. Let process p_i eventually deliver a sequence φ_k at a time t_k such that $w_k \in \varphi_k$ and $s \notin \varphi_k$. Assume that no event occurs at any process in Π_M after σ_{k-1} and before t_k. Assume that all messages in M_{k-1} sent by processes in Π_M (resp. Π_m) are received by processes in Π_m (resp. Π_M) in σ_k but after t_k . Let all enabled computation events occur. Finally, assume that all messages sent after σ_{k-1} are included in M_k and are not received in σ_k.

This proof first shows that the construction of σ_k is valid by showing that t_k and φ_k exist. It constructs an extension of σ_{k-1} called σ_{E1}. Assume that in σ_{E1} all processes in Π_M crash immediately after σ_{k-1} (i.e., before $submit_i(w_k)$) and $\Diamond\mathcal{S}$ suspects Π_M at all processes. Assume that all messages in M_{k-1} that are sent by processes in Π_M are lost. By property (P1), and since $\Diamond\mathcal{S}$ permanently satisfies weak accuracy, process p_i eventually delivers a sequence φ_k with $w_k \in \varphi_k$ at time t_k. Therefore, φ_k and t_k exist. As σ_{k-1} satisfies (I), process p_i cannot deliver s in σ_{E1} because all messages in M_{k-1} sent by processes in Π_M are lost. This implies that $s \notin \varphi_k$. Since process p_i cannot distinguish σ_k and σ_{E1} up to t_k, φ_k is delivered by p_i at time t_k in σ_k too.

The proof now shows the inductive step, i.e., that σ_k satisfies (I). Assume by contradiction that a sequence $\pi\, s\, \varepsilon_d$ for some sequences π and ε_d is delivered for the first time by a process p_d in σ_k or in an extension of σ_k respecting (i)-(ii). As s was not delivered in σ_{k-1}, sequence $\pi\, s\, \varepsilon_d$ is delivered after σ_{k-1} and, by the argument above, also after t_k.

Consider first the case $p_d \in \Pi_m$. Let σ_{E21} be an extension of σ_k where p_d delivers $\pi\, s\, \varepsilon_d$ and let t'_k be the time when this delivery occurs. Let all processes in Π_M crash immediately after σ_k and let all the messages sent by processes in Π_M sent after σ_{k-1} to processes in Π_m be lost. Finally, let $\Diamond\mathcal{S}$ return Π_M at all processes. From eventual stability and since p_i has already delivered at time $t_k < t'_k$ a sequence φ such that $w_k \in \varphi$ but $s \notin \varphi$, it follows $w_k \in \pi$.

Considers now a run σ_{E22} where the same events as in σ_{E21} occur until time t'_k but no process crashes before t'_k. All processes in Π_m crash immediately after t'_k. All messages sent from processes in Π_m to processes in Π_M after σ_{k-1} are lost. Assume that after t'_k, $\Diamond\mathcal{S}$ eventually returns Π_m at all processes in Π_M. p_d cannot distinguish σ_{E21} and σ_{E22} until t'_k, so it delivers $\pi\, s\, \varepsilon_d$ at time t'_k in σ_{E22} too. As all processes in Π_M are correct, they must eventually deliver a sequence containing s by (P2). From strong prefix consistency and strong prefix stability, this sequence must have $\pi\, s$ as prefix with $w_k \in \pi$.

Finally, consider a run σ_{E23} that is similar to σ_{E22} but where the *submit*$_i(w_k)$ event does not occur. Let all processes in Π_m crash at the same time as in σ_{E22}, and let all messages sent by processes in Π_m after σ_{k-1} be lost. Assume that no other process crashes. Let the outputs of $\Diamond\mathcal{S}$ be at any time the same as in σ_{E21}. Runs σ_{E21} and σ_{E22} are indistinguishable for the processes in Π_M, which thus eventually deliver a sequence having $\pi\, s$ as a prefix with $w_k \in \pi$. However, w_k has never been submitted in σ_{E23}. This violates nontriviality, showing that $p_d \notin \Pi_m$.

Next, consider the case $p_d \in \Pi_M$. By assumption, (I) holds so p_d must deliver $\pi\, s\, \varepsilon_d$ in σ_k. Let t''_k be the time when this occurs. Consider an extension σ_{E31} of σ_k where no process crashes. By (P2), all processes must eventually deliver a sequence containing s. By strong prefix consistency, all processes must eventually deliver a sequence having $\pi\, s$ as prefix. By eventual stability, since p_i has already delivered at time t_k a sequence φ_k including w_k and not s, it must hold $w_k \in \pi$. Before t''_k, process p_d cannot distinguish σ_k from a similar run σ_{E32} where *submit*$_i(w_k)$ does not occur. In fact, p_d does not receive any message before t''_k that is causally related with *submit*$_i(w_k)$. At time t''_k, therefore, p_d delivers $\pi\, s\, \varepsilon_d$ with $w_k \in \pi$ in σ_{E32} too, a violation of nontriviality. This ends the proof that σ_k satisfies (I).

The infinite run σ can be built iteratively by extending σ_k as it has been done with σ_{k-1}. The resulting run is fair by construction because all messages in M_{k-1} are delivered in σ_k and no computation event is enabled forever without occurring. During the whole run no process crashes. According to (P2), s should be delivered in a finite prefix of σ. By construction, however, each finite prefix τ of σ is also prefix of a run $\sigma_{k'}$ for some k'. From the invariant (I), s is never delivered in $\sigma_{k'}$, a contradiction. □

5.3.2 A Gracefully Degrading Implementation

In this section introduces Aurora, an algorithm implementing Eventual Consensus and thus, from Theorem 3, Eventual Linearizability. Aurora shows that Eventual Consensus can be implemented with any number of correct processes using $\Diamond\mathcal{S}$, still ensuring termination of weak operations and Eventual Consistency in worst-case asynchronous runs. The algorithm also shows that causal consistency can be combined with Eventual Consensus.

Failure detectors and communication primitives Aurora ensures termination of weak operations and Eventual Consistency in asynchronous runs. To this end, Aurora uses a failure detector module $\mathcal{D} \in \mathcal{C}$, which outputs the set of indices of the processes that have been suspected to crash. Virtually all failure detector implementations are of class $\mathcal{C}$ in asynchronous runs. The key property of Eventual Consensus, eventual stability, is achieved by letting a leader order all operations. For this, Aurora requires that $\mathcal{D} \in \Diamond\mathcal{S} \subseteq \mathcal{C}$, while for termination of strong operations it requires $\mathcal{D} \in \Diamond P \subseteq \Diamond\mathcal{S}$. This models the fact that even if Aurora optimistically relies on additional synchrony in order to achieve Eventual Consensus, the algorithm falls back to Eventual Consistency to ensure termination of weak operations in runs where Consensus would not terminate, including asynchronous runs. The use of $\Diamond\mathcal{P}$ to complete strong operations is a consequence of Theorem 4. For simplicity, $\Omega_{\mathcal{D}}$ is used to denote a simulation of a leader election oracle ensuring the properties of Ω on top of $\mathcal{D}$ in runs where $\mathcal{D} \in \Diamond\mathcal{S}$ similar to [Chu98]. The simulation ensures that the leader trusted by $\Omega_{\mathcal{D}}$ is not suspected by $\mathcal{D}$. The process that is permanently trusted by $\mathcal{D}$ when $\mathcal{D} \in \Omega_{\mathcal{D}}$ is called the *permanent leader*.

Processes use two communication primitives: a reliable channel providing *send* and *receive* primitives, and a (uniform) FIFO atomic broadcast primitive providing *abcast* and *abdeliver* primitives [AW04]. Implementing atomic broadcast is equivalent to solving consensus [CT96]. Aurora relies on an atomic broadcast implementations that use a failure detector Ω and a majority of correct processes for termination and that always respect their safety properties [Lam98; CT96]. The algorithm assumes that a predefined deterministic total order relationship $<_D$ exists. For simplicity, the algorithm sends and delivers whole histories although it is simple to optimize this away [FGL+96]. Garbage collection can be executed by periodically issuing strong operations for this purpose [SFK+09].

Properties of the Aurora algorithm Similar to weakly consistent implementations such as [LLSG92; TTP+95], Aurora ensures termination of weak operations, causal consistency and Eventual Consistency if $\mathcal{D} \in \mathcal{C}$. If $\mathcal{D} \in \Diamond\mathcal{S}$, Eventual Consensus is implemented. Termination of strong operations is ensured if $\mathcal{D} \in \Diamond\mathcal{P}$ or, in absence of concurrent weak operations, if $\mathcal{D} \in \Diamond\mathcal{S}$. All proofs are available in section C.

Checking if consensus will terminate A direct consequence of Theorem 4 is that if a leader p_{ld} has started consensus on a strong prefix $\pi\, s$ and it receives a weak operation w afterwards, it needs to distinguish whether consensus will terminate. If this is the case, w must wait to be ordered after $\pi\, s$ once consensus is reached. Else, w must be immediately be delivered since consensus will not terminate, and thus the strong operation will have to wait before being completed. Consensus will terminate if eventually there exists a stable majority of correct processes permanently trusting p_{ld}.[4]

Aurora uses *trust messages* to let p_{ld} know which processes trust it. Whenever $\Omega_{\mathcal{D}}$ outputs a new leader p_j at a process p_i, p_i sends a TRUST(j) message to all processes through FIFO reliable channels. Each process p_i keeps a *trusted-by set* TB including the indices of all the processes p_j such that TRUST(i) is the last trust message received by p_i from p_j. This processing of trust messages is not included in the pseudocode.

The leader uses the trusted-by set and a failure detector of class $\mathcal{C}$ to stop waiting for consensus unless consensus terminates. When a consensus instance is started, the leader remembers the subset T of TB that is composed only by correct processes (according to $\mathcal{D}$). Even in worst-case runs where $\mathcal{D} \in \mathcal{C}$, T will eventually include only correct processes. If T never changes and is a majority quorum, then there exists a majority of correct processes permanently trusting the leader. Consensus on $\pi\, s$ will thus eventually terminate, so the leader can wait to order and deliver w until this happens. The *wait-consensus* predicate is defined to reflect the aforementioned condition.

From Theorem 4, having a failure detector $\Diamond\mathcal{S}$, so a single leader, and a majority of correct processes is not sufficient to implement the properties of Aurora. The leader needs to eventually detect that such majority exists, which is ensured if $\mathcal{D} \in \Diamond\mathcal{P}$. This eventually lets the predicate *wait-consensus* be true whenever a consensus instance is ongoing, a sufficient condition for termination of strong operations. In fact, T will eventually be equal to the set of correct processes.

Note that if there is no concurrency between weak and strong operations, termination can be guaranteed for all operations without the need for distinguishing whether consensus can terminate.

Processing weak operations The processing of weak operations is described by Algorithm 10. When a weak operation o is submitted at a process p_i, p_i sends it in a *weak request* message to the current leader p_{ld} and waits for an answer from the leader. In order to preserve causal consistency, a weak request of p_i also contains its current history H and an associated round counter d which will be explained later. H contains all operations causally preceding o. When a weak request message m is received by p_{ld}, it merges its local history with the one received in m before adding o to its local history.

[4] A stable majority is defined as a majority quorum that does not change over time. The weakest failure detector to solve consensus, which is Ω, requires that eventually *all* correct processes permanently trust the same correct process p_{ld}. Appendix **??** shows that Ω can be simulated if eventually a stable majority of correct processes permanently trusts p_{ld}.

This is done in order to preserve causal consistency. The details of the merge operation (see Algorithm 11) will be discussed later on.

If the leader has proposed a strong prefix and is waiting to deliver it, it might wait until consensus on it is completed. This occurs if the leader thinks that consensus can be solved and therefore *wait-consensus* is true. In this case, the leader stores the request in the set W and waits until the strong prefix is delivered or *wait-consensus* becomes false. When p_{ld} processes the weak request, it sends a *push* message containing its local history, including also o, back to p_i. When p_i receives the push message, it merges the history of p_{ld} with its own history to order o respecting the causal dependencies of all the operations ordered by the leader before o. The resulting history contains o and is now delivered by p_i.

As already discussed, *wait-consensus* eventually becomes false unless consensus can be solved. Also, if p_{ld} is crashed, the failure detector will eventually suspect it. In the latter case, process p_i knows that no permanent leader is yet elected so eventual stability cannot yet be achieved. Therefore, p_i locally appends o to its current local history and delivers it without further waiting for a push message.

Processing strong operations - Overview The handling of strong operations is described by Algorithm 12 and is more complex. For eventual stability, if there is a permanent leader p_{ld} then strong operations should be delivered according to the order indicated by p_{ld}. However, the algorithm cannot rely on a leader to be permanent for strong prefix stability and consistency.

The properties of strong operations imply that delivering a strong prefix $\pi\, s$ requires solving consensus on $\pi\, s$. Equivalently, processes can propose strong prefixes by atomically broadcasting them and using some deterministic decision criteria to consistently choose one proposal. The main implication of Theorem 4, however, is that processes cannot just deliver the first strong prefix $\pi\, s$ proposed by a leader p_{ld}, even if this p_{ld} uses atomic broadcast. In fact, as long as p_{ld} believes that atomic broadcast will not terminate, it might have delivered some weak operation $w \not\in \pi$ before being able to abdeliver $\pi\, s$. In this case, p_{ld} cannot deliver $\pi\, s$ for eventual stability and it needs to propose a new prefix for s.

Processes need to decide when a proposed strong prefix can be delivered because it is *stable*, i.e. it has been abdelivered by atomic broadcast and no weak operation has been delivered in the meanwhile. Establishing that a prefix is stable is a local decision of a leader p_{ld}. The problem now is how p_{ld} can communicate this local decision and let other processes agree

Algorithm 10: Handling of weak operations

```
10.1  upon submit ($o$) and $o$ is weak
10.2      $ld \leftarrow \Omega_{\mathcal{D}}$;
10.3      send WREQ($H$, $d$ $op$) to $p_{ld}$;
10.4
10.5  upon receive WREQ($H'$, $d'$, $op'$) from $j$
10.6      if $wait\text{-}consensus$ and $(H', d', op') \notin W$ then
10.7          add ($H'$, $d'$, $op'$) into $W$;
10.8      else
10.9          $(H, d) \leftarrow$ merge($H'$, $d'$, $H$, $d$);
10.10         if $op' \notin H$ then append $op'$ onto $H$;
10.11         send PUSH($H$, $d$) to $p_j$;
10.12
10.13 upon receive PUSH($H'$, $d'$)
10.14     $(H, d) \leftarrow$ merge($H'$, $d'$, $H$, $d$);
10.15     deliver($H$);
10.16
10.17 upon $suspect\text{-}ld$
10.18     append last locally submitted weak operation onto $H$;
10.19     deliver($H$);
10.20
10.21 upon $stop\text{-}waiting\text{-}consensus$
10.22     foreach ($H'$, $d'$, $op'$) $\in W$ do
10.23         $(H, d) \leftarrow$ merge($H'$, $d'$, $H$, $d$);
10.24         if $op' \notin H$ then append $op'$ onto $H$;
10.25         send PUSH($H$, $d$) to $p_j$;
10.26         remove ($H'$, $d'$, $op'$) from $W$;
10.27
```

on its decision in presence of concurrent proposals from multiple leaders. If p_{ld} just atomically broadcasts that a prefix is stable, this creates again the same problem as before: all processes would have to wait that a stability confirmation from the leader is successfully broadcast before delivering the strong prefix. In the meanwhile, p_{ld} might locally store and deliver some new weak operation.

The problem of multiple concurrent leaders is solved in Aurora by using *rounds* and identifying a single leader as the *winner* of each round. Processes store the current round k and deliver a single strong prefix at each round. Leader processes that receive a new strong operation atomically broadcast the strong operation in a *proposal* message for the current round. The leader whose proposal is the first one to be atomically delivered for a round is the winner of that round. The winner of a round can propose multiple new strong prefixes for the round. These are received in the same order as they are abcast by the leader since the broadcast primitive is FIFO.

Assume that a proposed strong prefix becomes stable at the winner of the current round, that is, the winner abdelivers the stable prefix and sees that it is consistent with its current local history. The winner can now safely decide to locally store the strong prefix in its local history, deliver it, and

Algorithm 11: Background dissemination and merge

```
11.1  upon periodic tick
11.2      send PUSH($H$, $d$) to all other processes;
11.3
11.4  function merge($H'$, $d'$, $H$, $d$)
11.5      $d_{new} \leftarrow \max(d, d')$;
11.6      if $d = d_{new}$ then $H_{new} \leftarrow$ longest strong prefix of $H$;
11.7      else $H_{new} \leftarrow$ longest strong prefix of $H'$;
11.8      $O \leftarrow$ set of weak operations in $(H' \cup H) \setminus H_{new}$;
11.9      $R \leftarrow$ order $O$ according to $<_H \cup <_{H'}$ and break cycles according to $<_D$;
11.10     append $R$ onto $H_{new}$ in $R$ order;
11.11     return $(H_{new}, d_{new})$;
11.12
```

stop sending proposals for the round. The winner abcasts in this case a *close round* message indicating that the other processes can deliver its last proposed strong prefix for the round. A process abdelivering a close round message m for the current round delivers the last strong prefix proposed by the winner for that round and abdelivered before m. To ensure liveness in case a winner crashes, each process that suspects the winner of the current round can send a close round message.

Since proposal and close round messages are atomically broadcast, it is evident that all processes that did not win a round abdeliver the same strong prefix π for that round. Consistency with a winner of a round that has delivered a stable strong prefix based only on a local decision is ensured as follows. The prefix π is contained in the last proposal message m abdelivered by the winner, and thus by any other process, for the round, and it is not preceded by any close round message for the same round. Even if the winner crashes, all close round messages for the round will be abdelivered after m, ensuring consistency with the winner.

Eventually, only the permanent leader sends proposal and close round messages. This ensures that eventual stability is reached. Furthermore, if a majority is present in the system and $\mathcal{D} \in \Diamond\mathcal{P}$, eventually *wait-consensus* will be true during ongoing rounds of strong prefixes. This ensures that the leader eventually only adds weak operations between two rounds, ensuring termination of strong operations.

Processing strong operations - Detailed description In Algorithm 12, all processes keep two round counters: k stores the last round number of a proposed strong prefix, or the next round number if a prefix has just been delivered for a round; d denotes the highest round number for which a strong prefix has been stored in the local history. A submitted strong operation o is sent to all processes in a *strong request* message. When

Algorithm 12: Handling of strong operations

```
12.1  upon submit ($o$) and $o$ is strong
12.2      send SREQ($H$, $d$, $op$) to all processes;
12.3
12.4  upon receive SREQ($H'$, $d'$, $op$) from $j$
12.5      $(H, d) \leftarrow$ merge($H'$, $d'$, $H$, $d$);
12.6      add $op$ into $N$;
12.7
12.8  upon must-propose-new-prefix
12.9      $S \leftarrow N \setminus H$;
12.10     $Q \leftarrow H$;
12.11     $T \leftarrow TB \setminus \mathcal{D}$;
12.12     abcast PROP($Q$, $S$, $k$);
12.13
12.14 upon abdeliver PROP($H'$, $S$, $k'$) from $p_j$
12.15     if from-round-winner then
12.16         $P \leftarrow (H', S, k', j)$;
12.17     if proposal-stable then
12.18         foreach $op \in S$ in $<_D$ order do
12.19             append $op$ onto $H$;
12.20         $d \leftarrow k$;
12.21         deliver($H$);
12.22         abcast CLOSE-RND($k'$);
12.23
12.24 upon suspect-round-winner
12.25     abcast CLOSE-RND($k'$);
12.26
12.27 upon abdeliver CLOSE-RND($k'$) from $p_j$ and $P = (*, *, k', *)$
12.28     $P \leftarrow \perp$;
12.29     $Q \leftarrow \perp$;
12.30     $k \leftarrow k' + 1$;
12.31     let $H'$ and $S'$ be such that $P = (H', S', k', h)$;
12.32     $H_{new} \leftarrow H'$;
12.33     foreach $op \in S'$ in $<_D$ order do
12.34         append $op$ onto $H_{new}$;
12.35     $(H, d) \leftarrow$ merge($H_{new}$, $k'$, $H$, $d$);
12.36     deliver($H$);
12.37
```

a process receives such a message, it adds o to the set N containing all strong operations that have been received by the process.

If a process p_i believes to be a leader, it can make a proposal for a round if it has operations in N that have not yet been locally delivered and thus not yet inserted in the local history H. The sequence Q stores the last prefix that was proposed by p_i as a prefix of some new strong operation in the current round. A proposal is done by p_i only if p_i has not yet sent any proposal for the round, so $Q = \perp$,[5] or if a prefix has been proposed by p_i but some weak operations has been added to the local history H in the meanwhile so $H \neq Q$ (*must-propose-new-prefix* predicate). The proposal message contains H and the set $S = N \setminus H$ of new strong operations.

[5]The symbol $\perp$ denotes the value "undefined".

wait-consensus	$\triangleq$	$Q \neq \bot$ and $T = TB \setminus \mathcal{D}$ and $\|T\| > n/2$
suspect-ld	$\triangleq$	$ld \neq \Omega_{\mathcal{D}}$ and last locally submitted weak operation is not in H
stop-waiting-consensus	$\triangleq$	$W \neq \emptyset$ and $\neg$ *wait-consensus*
suspect-round-winner	$\triangleq$	$P = (*, *, k', j)$ and $j \neq \Omega_{\mathcal{D}}$
must-propose-new-prefix	$\triangleq$	$i = \Omega_{\mathcal{D}}$ and $N \setminus H \neq \emptyset$ and $(Q = \bot$ or $H \neq Q)$
from-round-winner	$\triangleq$	$(P = \bot$ and $k' = k)$ or $P = (*, *, k', j)$
proposal-stable	$\triangleq$	$j = i$ and $P = (*, *, k', i)$ and $H' = H$ and $k' = k > d$

Table 5.1: Predicates used by the Aurora protocol

If a new proposal message from the round winner is abdelivered, it is stored in the record P. If the winner decides that a proposal is stable, it stores it in H, delivers it, sends a close round message to all, and updates d. A close round message is also sent by any process that suspects the current round winner to be faulty. Whenever a close round message for the current round is received, the corresponding strong prefix is delivered. Before delivering a strong prefix, this is *merged* in the local history as described in Algorithm 11. The merge operation gives as result a history containing the strong prefix delivered in the largest round. All remaining weak operations are ordered after this prefix.

Background dissemination and merge In order to eventually converge to the same history, processes periodically send push messages to all other processes (Algorithm 11). The push mechanism is not only used to achieve Eventual Consistency. The permanent leader of a run uses push messages to fetch the histories of all processes and to aggregate them in a single consistent history. This is the key to achieve eventual stability. Strong prefix consistency and strong prefix stability are preserved by merges because, by construction, the longest strong prefix stored in a history H for round d is a prefix of the longest strong prefix stored in a history H' for round d' if $d \leq d'$. Causal consistency is preserved because all merged histories preserve it by construction. The merge only reorders operations that are ordered inconsistently in the two input histories. These operations, however, cannot be causally dependent. Inconsistent orderings of operations are eventually propagated to all processes and deterministically ordered using the $<_D$ relation. This is the key to eventual stability and consistency.

5.4 Chapter Summary

This chapter presented Eventual Linearizability and a related problem, Eventual Consensus. It established that combining Eventual Consensus with Consensus comes at the price of using a stronger failure detector than $\Diamond\mathcal{S}$, which is sufficient for Consensus. Finally, it presented Aurora, a gracefully-degrading shared object implementation extending Consensus with Eventual Consensus. Aurora only degrades consistency in periods when Consensus would block. It uses a failure detector of class $\Diamond\mathcal{P}$ to tell if Consensus will terminate, and one of class $\mathcal{C}$ to detect that Consensus will not terminate.

Chapter 6

Conclusions and Future Research

The thesis of this dissertation is that novel fault-tolerant replication algorithms are needed that fully adhere to the needs of Web-scale systems. In particular, the dissertation focuses on two main open issues. The first is reducing the performance and replication costs of tolerating worst-case failures, which are unlikely in general but do appear in very large-scale systems. The second is improving the efficiency of replication by increasing its availability, which has a positive impact both on latency and throughput, still keeping the same degree of consistency whenever possible. In this chapted we summarize the contributions of this thesis and indicate some new research directions for the future.

6.1 Overall Thesis Contributions

This section reviews the main contributions of this thesis and refers to the papers which have resulted from the thesis' work.

6.1.1 Low-Cost and Fast BFT

There has been a large deal of work on efficient and cheap BFT algorithms. The Scrooge algorithm represents a fresh look on existing lower bounds on the tradeoff between fast agreement and replication costs. The main idea is that a fast algorithm may not need to be always fast. By admitting a minor performance degradation upon failure events, Scrooge introduces a new upper bound on the replication cost of fast agreement. This is $2f + 2b$ replicas, where f is the overall number of tolerated faults (both crashes and Byzantine faults) and $b \leq f$ is the number of tolerated Byzantine faults. The existing lower bound for achieving fast agreement even in runs where a backup replica fails is $3f + 2b - 1$. This is $f + b - 2$ replicas more than the lower bound for Byzantine agreement, which is $2f + b + 1$. Scrooge thus shows for the first time that the additional costs to be fast in presence of faulty replicas is $f - 1$, that is, it is only a function of the number of the tolerated Byzantine faults. This makes Scrooge convenient in systems, like most Web-scale systems, where Byzantine faults are very rare.

Experimental evaluation shows that Scrooge performs as well as Zyzzyva and Zyzzyva5 in fault-free runs and that it performs like Zyzzyva5, and better than Zyzzyva, in runs with faults. These properties are achieved with strictly less replicas than Zyzzyva5. Scrooge also greatly outperforms Zyzzyva in presence of faults on read-only workloads.

The reduction of replication costs is particularly critical for Web-scale systems, which might include a large number of BFT clusters. A small reduction on the cost of a single cluster results in a significant reduction of hardware and energy costs if the number of clusters is high.

Resultant publication

- **Marco Serafini** and Neeraj Suri, *Reducing the Costs of Large-Scale BFT Replication*, in Proc. of Large-Scale Distributed Systems and Middleware (LADIS), 2008.

- **Marco Serafini**, Peter Bokor, Dan Dobre, Matthias Majuntke and Neeraj Suri, *Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas*, in Proc. of IEEE Int'l. Conf. on Dependable Systems and Networks (DSN-DCCS), 2010.

6.1.2 Fail-Heterogeneous Architectures

This thesis shows for the first time that trusted components can be used to reduce the replication costs of BFT even in general asynchronous systems. Relaxing the synchrony requirements compared to prior work on using trusted components is fundamental to enable the use of these components in Web-scale systems. The potential of this approach has been confirmed by the interest it has attracted. Immediately after HeterTrust, related and independently developed work has appeared. The A2M protocol aims at reducing replication costs by using an attested append-only memory [CMSK07]. This results in a symmetric failure mode for Byzantine processes, which resembles the hybrid fault model defined in [TP88]. Work on TRINC showed that such memory can be implemented using only a monotonically increasing counter which is associated with a key [LDLM09]. These papers propose using specialized hardware components that are deeply integrated into the processors' hardware. The fail-heterogeneous fault model does not impose such a restriction and is thus more generic. Trusted coordinators can be external processes with restricted software functionalities (only related to agreement) running on commodity hardware. However, the verification of trustworthiness for software processes is more complex than for hardware components.

The fail-heterogeneous architecture is also innovative in its use of trusted components as filters. While other work focuses only on integrity issues, HeterTrust shows that trusted components can also be used to preserve data confidentiality. Filtering in HeterTrust is not only done for confidentiality but also for tolerance to DoS attacks. Later work has shown that such filtering can also be done without assuming trusted components [ACKL08; CWA$^+$09], at the cost of degraded performance in "good" runs where no fault appears.

Resultant publication

- **Marco Serafini** and Neeraj Suri, *The Fail-Heterogeneous Architectural Model*, in Proc. of the IEEE Int'l Symp. on Reliable Distributed Systems (SRDS), 2007

6.1.3 Eventual Linearizability and Gracefully Degrading Implementations

Eventual Linearizability is a natural way of expressing gracefully degrading shared objects. In normal runs, these objects must respect the standard correctness condition of Linearizability. Whenever consistency deviates from Linearizability, it must eventually converge back to it.

This thesis introduced the first gracefully degrading replication algorithm, Aurora, which only relaxes Linearizability when a single leader is not available in the system. In such runs, consensus can not be solved and thus preserving Linearizability would mean blocking. Eventual Linearizability prevents blocking by relaxing consistency only in these cases. Aurora can be used to increase the consistency of existing weak consistency solutions for Web-scale systems without reducing availability.

It is often necessary to offer to applications the possibility of specifying different consistency degrees to different operations. Some operations may always require Linearizability, whereas other might be better off with Eventual Linearizability. The thesis shows that there are fundamental tradeoffs in combining Linearizability and Eventual Linearizability. In particular, strong operations can only be completed using a stronger failure detector than needed to solve Consensus.

A first investigation on the applicability of Eventual Linearizability to practical Web-scale applications, such as crawling, is in [SJ10]. These applications often partition their large workload over a large number of processors using master-worker schemes. Using Eventual Linearizability has the potential to be significantly more advantageous than using Linearizability in systems where partitions are not very rare.

Resultant publications

- **Marco Serafini**, Dan Dobre, Matthias Majuntke, Peter Bokor and Neeraj Suri, *Eventually Linearizable Shared Objects*, in Proc. of ACM Symp. on Principles of Distributed Computing (PODC), 2010.

6.2 Open Ends

Web-scale systems represent an evolving class of systems with mutating requirements. This thesis identifies some solutions but also opens up new research questions, as discussed in this section.

6.2.1 Negative Results

This thesis focuses on introducing positive, algorithmic results on the possibility of solving certain problems. The only negative results, which is the impossibility of section 5.3.1, was made necessary to justify the counterintuitive requirements of the Aurora protocol for completing strong operations. The positive results of this thesis call for further negative result that establish the necessity of some of the requirements of the proposed algorithms. There are two issues that are particularly interesting in this sense.

The Scrooge protocol shows that $2f + 2b$ replicas are sufficient for a BFT algorithm to be eventually fast in presence of faulty (unresponsive) replicas. It still remains to show a lower bounds matching this upper bound, that is, a proof that $2f + 2b$ replicas are minimal for being eventually fast.

Similarly, this thesis shows that a $\Diamond\mathcal{S}$ failure detector is *sufficient* to implement eventual linearizability. The open question is whether it is also necessary, that is, whether $\Diamond\mathcal{S}$ is the weakest failure detector which implements eventual linearizability.

6.2.2 Understanding Byzantine Faults

The design of BFT algorithms has reached a very mature state, which included the design of BFT versions of existing Web-scale services such as HDFS [CKL+09]. Assuming that the problem of designing efficient BFT systems can be solved, the main question that remains is whether using these systems is worth their additional complexity. In other words, it is not yet clear whether non-silent faults appearing in practice are best modeled using the Byzantine fault model [SJR09]. While in general the Byzantine fault model is attractive due to its generality, there still are a number of unresolved issues that are preliminary to the use of BFT.

First, many non-silent fault are caused by hardware malfunctions rather than malicious activity [Con02; Bor05; PWB07; SG07]. These faults do not require the use of cryptographic techniques. In fact, more efficient coding techniques can be used to detect errors induced by hardware faults. These do not only offer performance advantages, but also reduce the administrative complexity of setting up cryptographic algorithm, as for example generating and sharing secret keys.

Second, bugs and other software faults tend to be correlated, and may violate the assumption of failure independence. Design diversity is not proven to be effective and in many cases is not an option [LPS01]. Also, design diversity is unlikely to protect the system from configuration and maintenance faults, which often also have correlated nature [SJR09].

Third, using the Byzantine fault model for malicious intrusions still leaves many security issues open. Protecting confidentiality requires a very large number of nodes [YMV+03] unless, as this thesis shows, trusted components are used. Even with trusted components, however, confidentiality requires specific network topologies to prevent data leaks. It is not clear whether mandating the use of such topologies is realistic in data centers. Another issue is that BFT systems, like any other distributed system, are vulnerable to denial of service attacks. Although some solutions have been proposed to mitigate this problem, such as [ACKL08; CWA$^+$09], these again require the use of specific network topologies.

Overall, the application of BFT is limited by the lack of clear evidence that these faults occur in practice.

Research topics In the field of BFT replication, there are two main open issues that need to be solved to make a good case for the usefulness of this approach, also in the context of Web-scale systems. The first is establishing whether the Byzantine faults that appear in practical systems can be tolerated using BFT. The second and perhaps more important issue is whether it is possible to design algorithms that tolerate arbitrary but accidental faults (hardware) and that are more efficient and cheaper than BFT algorithms.

6.2.3 Applications of Eventual Linearizability

Evaluating the practical impact of Eventual Linearizability is also an open issue. Eventual Linearizability is useful for systems where weak consistency is acceptable but it is the last resort. These are systems where availability is of paramount importance, but the loss of consistency must be limited.

Follow-up work already made some evaluations related to the application of eventual linearizability in highly-available master-workers schemes [SJ10]. There are also other examples where eventual linearizability can be useful. Consider for example a system handling bids. Consider for example a bidding system. High availability is crucial to always ensure that users can do their bids, especially when the end of the bid is approaching. However, it is desirable that bidders can base their decisions on the most up-to-date information available. This is particularly true if the actual value of the bid is a function of the current state of the system. Another example where Eventual Linearizability is beneficial could be a flight booking system. Ensuring high availability is essential avoid keeping seats unsold. Relaxing consistency to preserve high availability might result in over-bookings, that are anyway tolerated by the application. However, in the normal case, updated real-time information should be provided to each retailer and customer. A final example could be a social networking application. In order to increase data locality, user accounts can be spread over a wide-area system. Each user might have friends worldwide who want to observe updates to its profile and to comment them. It is desirable that, when multiple users are concurrently commenting on a friend's picture or on a status update, they observe a real-time flow of comments. Degradation of consistency, however, is preferable to unavailability, which can demotivate the user from interacting with the system.

Existing weakly consistent solutions have some limitations which are a direct consequence of their consistency semantics. First, they either always degrade to causal consistency, as Dynamo [DHJ$^+$07] or have stronger consistency but become unavailable if one of the replicas, identified as the master, becomes unavailable, as for example PNUTS [CRS$^+$08]. Eventual Linearizability provides strong consistency most of the time, and degrades consistency only when implementing Linearizability would mean blocking the system. It offers the same advantages as eventually consistent systems in terms of availability, but it only allows divergences when it is necessary.

Research topics Evaluating Eventual Linearizability in practical applications scenarios could be done on multiple dimensions. Existing work shows some initial results indicating that it can be beneficial in highly available master-worker schemes when the likelihood of partitions or timing failures using strong consistency is not negligible [SJ10]. Eventual Linearizability seems also to be attractive for other applications, such as bidding, retail or social networking applications, where the loss of consistency is acceptable but not desirable and should be minimized. This claim must be validated considering some specific use case.

Appendix A

Scrooge

This section presents additional results related to the Scrooge protocol. Section A.1 shows the correctness of the algorithm. Section A.2 presents an extension to Scrooge enabling garbage collection and shows it correct.

A.1 Correctness of the Scrooge Protocol

This section proves the correctness of the simplified Scrooge protocol. The next two sections describe the full version of Scrooge by extending the simplified version, and prove that the introduced modifications preserve correctness. As customary, first it is proven that the protocol never violates some invariant properties (safety) and that the protocol eventually achieves some useful results (liveness).

A.1.1 Replica State and Definitions

Consider systems composed by $N \geq 2f + 2b$ replicas. At most $f > 0$ replicas can be faulty and at most b can be Byzantine, with $0 < b \leq f$, while the others only crash.

Proofs reason in terms of the *message history*, or simply history, stored by replicas. A history is an array indexed by a unique *sequence number* n. Each history element is a triple including the following fields:

- a client request m
- a replier quorum RQ
- a primary authenticator μ_p

Replicas go through a sequence of views, and accept messages of the agreement protocol for a view $v > 0$ only if after the view change to v is completed. In this case the view is called is *established.* A replica is *in view* v if v is its last established view. If a correct replica i in view v participates to a view change to a view $v' > v$, it can build a *tentative history* for v', which is denoted as *t-hist(v, i).* Tentative histories are indicated by the primary of the new view v'. When a correct replica i establishes view v', it agrees with the other replicas on the tentative history (Lines 2.30 – 2.34), which then becomes an *established history* and is denoted as *e-hist(v, i).* Each view v has a pre-defined primary $p(v)$. In the pseudocode, both tentative and established histories are denoted by ih during view change.

A *history prefix in view v for sequence number n and correct replica i* is denoted as *prefix(n, v, i)* and is defined as the subset of the message history elements stored in view v by i with sequence number $n' \leq n$. Given two histories h and h', *h is a prefix of h'* iff for each element of h there is also one identical element of h' associated with the same sequence number. Furthermore, h is a *request prefix of* h' iff for each history element of h there is an element in h' associated with the same sequence number, client request and replier quorum.

An *agreed history prefix in view v for sequence number n and correct replica i* is any history prefix in view v for n and i where $n \leq aw$ and aw is the agreement watermark of i at the end of view v. It is denoted as *a-prefix(n, v, i).* Committed history prefixes *c-prefix(n, v, i)* are defined similarly.

View establishment certificates are attached to each view change message. A view establishment certificate E received from a replica i is *valid for a view change message vc* if $E = vc.E$ and all its $N - f$ signed establish view messages ev from different replicas contain the same view number $ev.v = vc.v$, the same sequence number $ev.n \leq vc.aw$ and the same correct digest $ev.h$ of the history prefix $\{vc.mh[0], \ldots, vc.mh[ev.n]\}$.

A replier quorum $RQ' \neq \perp$ is said to be *valid for correct replica i in view v at sequence number n* if, given the highest sequence number $n_c \leq n$ such that *c-prefix(n_c, v, i)*, all the history elements with sequence number in $[n_c, n]$ contain RQ'. This is denoted as *RQ-valid(RQ', n, v, i).* A replier quorum $RQ' \neq \perp$ is said to be *current for correct replica i in view v at request n* if replica i in view v has set RQ to RQ' when it handles the order request for sequence number n in Lines 1.15 – 1.26. This is denoted as *RQ-current(RQ', n, v, i).* The difference between the two predicates is that the first refers to replier quorums stored in the history logged by a replica, while the second refers to the current replier quorum of the replica when an order request message is processed.

A correct client c completes an operation o after receiving replies from

Name	Description	Type	Init
v	current view	timestamp	0
RQ	replier quorum	set of pids	[0,N-f-1]
n	current seq. number	timestamp	0
mh	message history	array of $\langle$*req.*,RQ,*auth.*$\rangle$	$\perp$
h	history digests	array of digests	$\perp$
aw	agreed watermark	timestamp	0
cw	commit watermark	timestamp	0
SL	suspect list	set of f pids	[N-f,N-1]
n_{SL}	seq. number of SL	timestamp	0
v'	new view	timestamp	0
ih	initial history	array of $\langle m, RQ,$ *auth.*$\rangle$	$\perp$
E	view establishment certificate	set of $N-f$ signed EST-VIEW messages	default value

Table A.1: Global variables of a replica

a quorum of replicas in view v which executed the request with the same sequence number n and history prefix h. This is denoted as *complete(o, n, v, h, c)*.

A.1.2 Agreement and Helper Procedures

The pseudocode of the reconfiguration phase is Algorithm 13, the one of the helper procedures is Algorithm 14, and the list of variables of the algorithm is in Table A.1.

A.1.3 Proof Sketch

This section provides proof sketches for the safety and liveness properties of the protocol. For simplicity, it is assumed in the proof sketch that $f = b$.

Safety

The safety property of BFT replication protocols is that clients have the abstraction of interacting with a non-replicated server executing all requests according to a total order [Sch90]. Therefore, if two clients issue two different requests and complete them, these must have been consistently ordered by the replicas which have generated the delivered replies. The following argument argues that this property holds within a view and across views.

Within a view: A client can complete requests in a view in two cases: either after receiving $3f$ matching speculative replies or after receiving $f+1$ stable replies. If two clients complete a request, they receive speculative replies from two sets of replicas which intersect in one correct replica i. Since correct replicas execute requests in the order dictated by sequence numbers, replica i has established a local order between the two requests. A client completes a request from speculative replies only if $3f$ repliers have sent a consistent history digest. This implies that these replicas have the same history as i, so the requests are consistently ordered. A similar reasoning can be done if one of the client, or both, deliver after receiving stable replies. In fact, stable replies are agreed by replicas after receiving a set of $3f$ agreement messages with matching histories and any two such set intersect in a correct replica i.

Across views: If a client completes a request m after receiving replies for view v and sequence number n and a view change to view $v+1$ occurs afterwards, m must be associated with n in the new view as well. This ensures that if a second client completes a request in $v+1$, and by induction in any successive view too, the two requests will be consistently ordered. Consider now consider how m is recovered during view change.

If m has completed from $f+1$ stable replies, at least one correct replica has received in view v matching commit messages for n from at least $2f$ correct replicas. These replicas have a common agreed history prefix including all elements with sequence numbers $n' \leq n$. During the view change protocol, at least f of these correct replicas will send a view change message to the new primary. Request m is thus included in one agreed candidate. However, at most f other faulty replicas might report a different agreed candidate. In order not to be discarded, agreed candidates must be contained in the history of at least $f+1$ replicas, one of which is correct. If this holds for both candidates, it implies that the primary has sent two different ordered request messages for n and is thus one of the f replicas reporting a different candidate. The view change message from the primary is discarded in this case and a view change message from a correct replica, which can only report m as agreed candidate for n, is awaited. After that, only the agreed candidate containing m remains and is selected by the recovery function.

If m has completed after the client has received $3f$ speculative replies from a replier quorum RQ, at least $2f$ correct repliers i had updated their current quorum RQ_i to RQ when they had committed on an agreement watermark $n_a^i < n$. The history element for n_a^i, and thus the replier quorum RQ contained in it, can thus be recovered. Before delivering, the client has also checked that all the $2f$ correct repliers i have a common history consisting of elements with sequence numbers $n' \leq n$. Also, as the request has been speculatively replied by all correct replicas in RQ, the SPEC-RUN predicate was true when the repliers received the order request message for n. This implies that RQ_i was not set to $\perp$ by any of the correct repliers. The replier quorum RQ is thus contained in all history elements stored with sequence number n' such that $n_a^i \leq n' \leq n$ by each correct replier. By induction on n', RQ_{n-1} is set equal to RQ when a agreement is reached for n. During view change, the primary will receive at least f view change messages from elements in RQ_{n-1} associating m to n. The request m is thus associated with an ordered candidate, which is selected for sequence number n by the recovery function and included into the initial history of v' as discussed in section 3.4.2. If a different request m' is reported as agreed candidate by f faulty replicas and is included in the histories of $f + 1$ view change messages, this indicates that the old primary of view v was faulty and has sent inconsistent ordered requests. Its message is thus discarded and another message from a correct replica is awaited. After this is received, m' is not an agreed candidate any longer as correct replicas can only agree on m.

Liveness

Liveness is guaranteed when the system is in timely periods and thus the same view can be established by all correct replicas. If a client c cannot complete its request m from speculative replies, it resends m to all replicas. If the primary is correct, the SPEC-RUN condition ensures that the agreement and commit phases are executed by each correct replica once one correct replica receives both m from c and the corresponding order request message from the primary. All correct replicas initiate and complete the agreement and commit phases on the entire history prior to m and send to $f+1$ matching stable replies to c. These are sufficient to complete the request. If this does not eventually happen, the primary is faulty and at least $f+1$ correct replicas accuse it. This is sufficient to let all replicas initiate a view change.

The protocol cannot block during view change in timely periods. A new correct primary can always wait for $3f$ well-formed view change messages from correct replicas. Each of them eventually becomes stable as the outcome vectors o contain binary boolean values and the new primary receives outcome vectors from $3f$ correct replicas for each of these view change messages. Additional messages are waited if there are multiple candidates for a sequence number and one of them is included in the view change message sent by the primary of the last established view v. This implies that at least one faulty replica has sent a view change message and is thus possible to wait for one additional view change message. Also, if an agreed candidate e is sent by correct replicas, at least $2f$ correct replicas have it in their local history. If an incorrect agreed candidate is reported by a faulty replica, all $3f$ correct replicas send view change messages reporting that no agreement on it was reached. In both cases progress is ensured.

A.1.4 Scrooge Safety

The safety property provided by BFT replication protocols is that different correct clients always observe consistent histories, as reflected in the following *safety property.*

Property 1. *For each pair of correct clients c_1 and c_2 and for each pair of operations o_1 and o_2 completed by client c_1 and c_2 respectively, let n_1 and n_2 be the sequence numbers associated with o_1 and o_2 respectively and h_1 and h_2 the history prefixes stored in any view by any correct replica for n_1 and n_2 respectively. If $n_1 \leq n_2$, then h_1 is a request prefix of h_2.*

The purpose of this section is to prove that Property 1 is an invariant.

First some consistency properties within a view v are proven, and then it is shown how consistency is preserved across views.

Lemma 1. If $h =$ *e-hist(v, i)* and $h' =$ *e-hist(v, j)*, then $h = h'$.

Proof: If $v = 0$ all replicas have the same established initial history, which is empty.

If $v > 0$, replicas consider their initial history for view v as established in Lines 2.30 – 2.34 after having received valid establish view messages with matching history digests from a quorum of $N - f$ replicas, including at least $N - f - b \geq f + b$ correct replicas. If two correct replicas i and j have established the same view v, there are at least $b > 0$ correct replicas k in the intersection of the quorums of i and j. Since a correct replica accepts only one new view message per view in Lines 2.24 – 2.28, it never sends two different establish view message. The initial histories for i and j are thus the same.

Lemma 2. If $h =$ *prefix(n, v, i)* and $h' =$ *prefix(n', v, i)* and $n \leq n'$ then h is a prefix of h'.

Proof: A correct replica i in view v adds an entry to its history only upon receiving order request messages from the current primary $p(v)$ (Lines 1.15 – 1.26). The entry for sequence number n is added only if n is the smallest sequence number not yet associated with an entry in the history of i. This implies that (a) only one entry can be associated with a given sequence number in a history in view v, (b) there are no gaps and (c) if h is the history prefix for sequence number n, requests for higher sequence numbers n' added in view v are appended to h.

Lemma 3. If *RQ-current(Q, n, v, i)* then *RQ-valid(Q, n, v, i).*

Proof: Let n_c be the highest sequence number smaller than n such that *c-prefix(n_c, v, i)* and assume by contradiction that *RQ-valid(Q, $n-1$, v, i)* does not hold. This implies that some history element with sequence number in $[n_c, n-1]$ contains a replier quorum $S \neq Q$. Let $n_S \geq n_c$ be the highest sequence number of such an element. From *RQ-current(Q, n, v, i)* it follows that $RQ = Q$ for i when the order request with sequence number n is processed by i in view v (Lines 1.15 – 1.26). Order requests are processed following their sequence numbers and the replier quorum RQ is set to $\perp$ if the predicate SPEC-RUN does not hold. RQ is set to a value $Q \neq \perp$ in view v only when a new commit watermark is reached in v and all history elements from the commit watermark up to the current sequence number are associated with Q in the message history of i (Lines 13.30 – 13.40). Therefore, *RQ-current(Q, n, v, i)* implies that (a) there exists a sequence number $n_Q < n$ such that a commit on n_Q is reached in view v before an order request with sequence number $k < n$ is processed and Q is associated to all history elements in $[n_Q, k-1]$, and (b) RQ is not set to $\perp$ when order request with sequence numbers in $[k, n]$ are processed, so SPEC-RUN holds and $RQ_p = Q$ for all the corresponding history elements. This implies that $n_Q > n_S \geq n_c$, which contradicts the definition of n_c.

Lemma 4. If a correct replica i in view v sends a speculative reply for a request associated with sequence number n in its history, then there exists a replier quorum Q such that *RQ-valid(Q, n, v, i)* and $i \in Q$.

Proof: If i sends the speculative reply in Lines 1.15 – 1.26 it follows from SPEC-RUN that it received an order request message for n from the primary $p(v)$ containing a replier quorum $RQ \neq \perp$ such that *RQ-current(RQ, n, v, i)* holds and $i \in Q$. The result thus follows from Lemma 3.

If i sends the speculative reply after a commit in Line 13.40, the result follows from the fact that the procedure *send-missing-spec-rep* checks that $i \in Q$ and that Q is associated to the history element with the highest committed sequence number $n_c \leq n$ and to all history elements with sequence numbers in $[n_c, n]$. This ensures that each commit with sequence number in $[n_c, n]$ is associated with Q, as required by *RQ-valid(Q, n, v, i)*.

Lemma 5. If $h =$ *a-prefix(n, v, i)* then either h is a prefix of *e-hist(v, i)* or there exist $N - f - b$ correct replicas j such that $h =$ *prefix(n, v, j)*.

Proof: A correct replica i can update its agreed watermark to $aw \geq n$ in two cases: when it establishes the view v in Lines 2.30 – 2.34 or when it completes an agreement phase for sequence number aw in Lines 13.23 – 13.28. In the first case, h is a prefix of *e-hist(v, i)* as from Lines 2.30 – 2.34.

In the second case, replica i has received equal agree messages containing a history digest for h and sequence number n from a quorum Q of $N - f$ replicas in view v. At least $N - f - b$ replicas in Q are correct and have thus sent matching agree messages only if h = *prefix(n, v, j)*.

Lemma 6. If a_i = *a-prefix(n_i, v, i)* and a_j = *a-prefix(n_j, v, j)* and $n_i \leq n_j$, then a_i is a prefix of a_j.

Proof: From a_i = *a-prefix(n_i, v, i)* and Lemma 5 either a_i is a prefix of *e-hist(v, i)* or there exist at least $N - f - b$ correct replicas k such that a_i = *prefix(n_i, v, k)*.

In the first case, it follows that if any correct replica j has a history prefix a_j in view v, then *e-hist(v, j)* is a prefix of a_j (from Lemma 1) and *e-hist(v, j)* = *e-hist(v, i)* (from Lemma 2). It thus follows that a_i is a prefix of a_j.

In the second case, it follows from Lemma 5 that there exists a set of at least $N - f - b \geq f + b$ correct replicas k such that a_i = *prefix(n_i, v, k)*. As $n_j \geq n_i$, replica j also sets its agreement watermark in Lines 13.23 – 13.28 after receiving agree messages from at least $b > 0$ of these correct replicas k reporting that $a_j = a_k$ = *prefix(n_j, v, k)*. Since k is correct, from Lemma 2 and $n_j \geq n_i$ it follows that *prefix(n_i, v, k)* is a prefix of *prefix(n_j, v, k)*, and thus a_i is a prefix of a_j.

Lemma 7. If h = *a-prefix(n, v, i)* and there exist at least $f + 1$ correct replicas j and a history prefix h' = *prefix(n', v, j)* for $n' \geq n$, then h is a prefix of h'.

Proof: From h = *a-prefix(n, v, i)* and Lemma 5 either h is a prefix of *e-hist(v, i)* or there exist $N - f - b$ correct replicas k such that h = *prefix(n, v, k)*.

In the first case, it follows that if any correct replica j has a history prefix h' in view v, then *e-hist(v, j)* is a prefix of h' (from Lemma 1) and *e-hist(v, j)* = *e-hist(v, i)* (from Lemma 2). It thus follows that h is a prefix of h'.

In the second case, a set S of at least $N - f - b$ correct replicas k have h = *prefix(n, v, k)*. From Lemma 2 it follows that if any of these replicas has an history prefix h'' = *prefix(n', v, k)* for $n' \geq n$, then h is a prefix of h''. Since each set of $f + 1$ correct replicas j intersect with one correct replica in S, their common history prefix h' is equal to h''.

It is now possible to show how the protocol preserves consistency across views. The following lemmas are the core lemmas to prove the safety of the protocol.

Lemma 8. If ih = *t-hist*($v+1$, *i*) and there exist a sequence number n and $N-f$ replicas j such that *a-prefix*(n_j, *v*, *j*) for $n_j \geq n$ if j is correct, then all j have the same history prefix hp = *prefix*(*n*, *v*, *j*) and hp is a request prefix of ih.

Proof: Consider the case where i is the primary of view $v+1$. The case of the other backup replicas is similar since the same decision procedure *recover* used by the primary to recover ih is used by the backups.

In Lines 2.7 – 2.18 of the view change to view $v+1$, the primary of the new view $v+1$ receives view change messages vc with message histories $vc.mh$ including h' as a prefix and with $vc.aw \geq n$ from at least $N-2f-b$ of the at least $N-f-b$ correct replicas j. Since v is the highest established view smaller than $v+1$ and since it is contained in $N-2f-b \geq b > 0$ view change messages in VC, v is selected as the highest previous established view mv in Lines 3.11 – 3.15. From Lemmas 1 and 2, *e-hist*(*v*, *i*) is a prefix of hp. Also, *e-hist*(*v*, *i*) is a prefix of ih because $mv = v$ implies that the initial history ih is set to *e-hist*(*v*, *i*) in Lines 3.11 – 3.15. The proof first shows that the suffix of hp which is not included in *e-hist*(*v*, *i*) is also included in ih in Lines 3.16 – 3.31.

From the hypothesis, there exist at most f correct replicas l such that $h' \neq$ *a-prefix*(*n*, *v*, *l*). For each history element e with sequence number $n' \leq n$ in the suffix of hp, AGREED-CAND(e, n', v, VC) holds. In fact, apart at most b Byzantine replicas and f correct replicas, all other correct replicas j send view change messages vc to the new primary of v' such that $vc.v = mv = v$, $e = vc.mh[n']$ and $vc.aw \geq n$. This is because for all replicas j, hp = *a-prefix*(*n*, *v*, *j*) as $n_j \geq n$ for all j and from Lemma 6. It is now shown that e is the only possible agreed candidate which is selected for n' in ih in Lines 3.23 – 3.24.

Assume by contradiction that a candidate $g \neq e$ is selected for n'. As e is an agreed candidate, g must be an agreed candidate which is selected in Lines 3.23 – 3.24 and predicates WAIT-AGR(A, n', v, VC) and WAIT-ORD(A, O, n', v, VC) are false. As g is an agreed candidate, $b+1$ replicas, including at least a correct one, have received from the primary of view v a different order request for n' than the replicas which agreed on e. The old primary $p(v)$ of view v is thus Byzantine. From Lemma 6, no correct replica can send a view change message with g in its agreed history prefix for n'. By definition of AGREED-CAND, g is an agreed candidate only if all $|VC| - f - b \geq b$ Byzantine replicas, including the primary, have sent a view change message with g in the agreed history prefix for n' and v and all these messages are in VC. As WAIT-ORD is false and there are multiple different candidates, the new primary of view $v+1$ waits until $|VC| > N-f$.

This implies that g is not selected as an agreed candidate as g is included in the agreed prefix of view change messages in VC sent by at most b replicas but $|VC| - f - b > N - 2f - b \geq b$.

Lemma 9. If ih = *t-hist(*$v + 1$*, i)* and there exist a history prefix *hp* and a replier quorum *RQ* such that $|RQ| = N - f$ and $Q \subseteq RQ$ is the subset of all correct replicas in *RQ* and for each $j \in Q$, *hp = prefix(n, v, j)* and *RQ-valid(RQ, n, v, j)*, then *hp* is a request prefix of *ih*.

Proof: Consider the case where i is the primary of view $v + 1$. The case of the other backup replicas is similar since the same *recover* function used by the primary to recover *ih* is used the the backups.

Let l be the replica having the the smallest agreement watermark aw_l among the replicas in Q. As Q contains $N - f - b$ correct replicas, Lemma 8 implies that the history prefix h' = *prefix(*aw_l*, v, l)* is a prefix of the initial history *ih*. If $n \leq aw_l$, then it follows from Lemma 2 that *hp* is a prefix of h', q.e.d. If $n > aw_l$ then *RQ-valid(RQ, n, v, l)* implies by definition that *RQ* is contained in the history element s of h' for aw_l, which is the highest agreement watermark $\leq n$ of replica l. It thus follows from Lemma 8 that *RQ* is contained in the candidate which is selected for sequence number aw_l and subsequently used to identify the observed candidates for sequence number $aw_l + 1$. The next step is proving by induction that for all sequence numbers n' such that $aw_l < n' \leq n$, the client request included in *hp* for n' is selected for the initial history *ih*.

The inductive hypothesis implies that $RQ_{n'-1} = RQ$. As $Q \subseteq RQ$ contains at least $N - f - b$ correct replicas, the new primary of view $v+1$ receives view change messages from at least b of them. All these replicas report the same history element for n', which is thus a candidate e. The replier quorum *RQ* is included in e since for all $j \in Q$ it holds that *RQ-valid(RQ, n, v, j)*. Assume by contradiction that a candidate $g \neq e$ is selected for n'. The candidate g must be selected in one of the three cases of Lines 3.23 – 3.28. The following shows that in each of these cases a contradiction is reached.

If g is selected in Lines 3.23 – 3.24, this implies that g is an agreed candidate, AGREED-CAND(g, n', v, VC) holds and the predicates WAIT-AGR(A, n', v, VC) and WAIT-ORD(A, O, n', v, VC) are false. As AGREED-CAND holds, g is included in the local history of $b+1$ replicas, including a correct one. This implies that g, as well as e, has been associated by the old primary $p(v)$ to sequence number n. The old primary $p(v)$ has thus sent inconsistent order request messages for n' and is thus Byzantine. From Lemma 7, as Q contains $N - f - b \geq f + b > f$ replicas, only Byzantine replicas can claim to have agreed on g for n' and view v. It follows from

AGREED-CAND that all $|VC| - f - b \geq b$ Byzantine replicas, including the primary, have included g in the agreed history prefixes of their view change messages, and all these view change messages are included in VC. If g is selected then WAIT-ORD is false and $|VC| > N - f$ as there are two different candidates. In order to be selected as an agreed candidate, g there must be at least one correct replica which has agreed on g. This contradicts Lemma 7 as Q contains more than f correct replicas.

If g is an observed candidate selected in Lines 3.25 – 3.28, either g satisfies VERIFIED(g, VC, CH) because at least one correct replica was able to verify that the corresponding order request message was generated from the old primary $p(v)$, or e does not satisfy VERIFIED(e, VC, CH). This in turns implies that $p(v)$ is Byzantine. In fact, if $p(v)$ were correct then $p(v) \in Q$, since *RQ-valid(RQ, n', v, j)* holds for some correct replica j which always checks that the primary of a view is a member of the replier quorums in that view. All replicas $j \in Q$ would have the same history prefix hp = *prefix(n, v, j)* as $p(v)$, including the same authenticator which was generated by $p(v)$ for the order request message corresponding to e. The candidate e, which would be the only one generated by primary $p(v)$, would thus be the only verified candidate, a contradiction.

By hypothesis, only faulty replicas in $Q = RQ_{n'-1}$ can associate g to n'. In order to be a candidate, g must be associated to n' in the view change messages sent to the new primary of v by all the $|VC| - f - b \geq b$ Byzantine replicas, including at the old primary $p(v)$, and all these messages must be included in VC. As there are multiple candidates and WAIT-ORD must be false for a candidate to be selected, $|VC| > N - f$. From the definition of ORDERED-CAND, g must be contained in the local history of at least $|VC| - f - b > b$ replicas in RQ_{n-1}. As there are at most b Byzantine replicas, g is associated to n' in the message history of at least one correct replica in RQ_{n-1}, which is also included in Q by definition. This contradicts the fact that $e \neq g$ corresponds to a common history element for n' and for all replicas $j \in Q$.

Lemma 10. If complete(o, n, v, h, c) and h' = *t-hist($v+1$, i)* then h is a request prefix of h'.

Proof: A client completes the request m in Lines 1.28 – 1.30 or in Lines 13.13 – 13.14.

If the client completes o in Lines 13.13 – 13.14, it has received $b+1$ stable replies from correct replicas j whose committed prefixes include h as a prefix and whose commit watermarks are $n_j \geq n$. This implies that at least one correct replica has received in Lines 13.30 – 13.40 consistent agree messages for its agreed history prefix from at least $N - f$ replicas. From Lemma 8 it follows that for each correct replica i, h is a prefix of h'.

If the client completes o in Lines 1.28 – 1.304, it has received speculative replies from a set RQ of $3f$ replicas j claiming to have the same history prefix $h =$ *prefix (n, v, j)* and to be members of the same replier quorum RQ such that $p(v) \in RQ$. Let Q be the subset of correct replicas in RQ. From Lemma 4, for each $j \in Q$, *RQ-valid(RQ, n, v, j)* and $j \in RQ$. Therefore, $Q \subseteq RQ$. From Lemma 9, it follows that for each correct replica i, h is a prefix of h'.

Lemma 11. If complete(o, n, v, h, c) and $h' =$ *t-hist(v', i)* and $v < v'$ then h is a request prefix of h'.

Proof: Assume by contradiction that h is not a prefix of h'. If no correct replica had established a view v'' such that $v < v'' < v'$, then all correct replicas would send for view v' the same view change messages as the ones sent for view $v + 1$ except from the new view field. A contradiction would thus follow from Lemma 10. Therefore, the primary of view v' must have received view change messages from some replicas j having a valid view establishment certificate for an established view v_j with $v < v_j < v'$ and for a corresponding established history $h_j =$ *e-hist(*v_j*, j)* $=$ *t-hist(*v_j*, j)*. Let k be, among the replicas j, the replica which sends the view change message with the highest established view v_k. This implies that h_k is selected as initial history ih_{v_k} by the recover function. From *complete(o, n, v, h, c)* and $v < v_k$ it follows that if h is not a prefix of h', then h is not a prefix of $h_k =$ *t-hist(*v_j*, j)*. This argument for v' can be inductively be applied to v_k. By induction on the largest established view $v'' < v'$ reported to the new primary of view v'', h is not a prefix of *t-hist(*v''*, i)*. Let v_i be the smallest view $v'' > v$ established by any correct replica i. All correct replicas send for view v_i the same view change messages as the ones sent for view $v + 1$ except from the new view field, but h is not a prefix of *t-hist(*v_i*, i)*. This contradicts Lemma 10.

Lemma 12. If complete(o, n, v, h, c) and complete(o', n', v, h', c') and $n \leq n'$ then h is a request prefix of h'.

Proof: Two clients c and c' can complete a request either in Lines 1.28 – 1.30 or in Lines 13.13 – 13.14. If client c completes a request after receiving $b + 1$ stable replies in Lines 13.13 – 13.14, then $h =$ *a-prefix(n, v, i)* for at least $N - f - b$ correct replicas i. If client c' completes a request after receiving $b + 1$ stable replies in Lines 13.13 – 13.14, then $h' =$ *a-prefix(n', v, j)* for at least $N - f - b$ correct replicas j. From Lemma 6 and $n \leq n'$ it follows that h is a prefix of h'. If client c' delivers from $3f$ speculative replies in Lines 1.28 – 1.30, then h' = *prefix(*n'*, v, j)* for at least $N - f - b$ correct replicas j. From Lemma 7, it follows that h is a prefix of h.

If client c completes after receiving $N-f$ speculative replies in Lines 1.28 – 1.30, then h = *prefix(n, v, i)* for a set Q of at least $N - f - b$ correct replicas i. In order to deliver either Lines 1.28 – 1.30, client c' must receive one reply from at least one correct replica $i \in Q$, and the result follows from Lemma 2. If c' completes a request after receiving $b + 1$ stable replies in Lines 13.13 – 13.14, then h' = *a-prefix(*n'*, v, j)* for at least $N - f - b$ correct replicas j, including at least one replica in Q. From Lemma 2, this implies that h is a prefix of h'.

Lemma 13. If complete(o, n, v, h, c) and complete(o', n', v', h', c') and $n \leq n'$ and $v < v'$, then h is a request prefix of h'.

Proof: If client c' completes a request in view v', this implies that it receives speculative or stable replies from at least one correct replica i in view v' and that h' = *prefix(*n'*,* v'*, i)*. Since this replica has established v', there exists an established history h'' = *e-hist(*v'*, i)* = *t-hist(*v'*, i)* = *prefix(*n''*,* v'*, i)*. From *complete(o, n, v, h, c)*, $v < v'$ and Lemma 11, h is a prefix of h''. It follows that h = *prefix(n,* v'*, i)* and thus that h, h' and h'' are all prefixes of i in view v'. As $n \leq n'$, h is a prefix of h' from Lemma 2.

Lemma 14. If complete(o, n, v, h, c) and complete(o', n', v', h', c') and $n \leq n'$ and $v > v'$, then h is a request prefix of h'.

Proof: If client c completes a request in view v, this implies that it receives speculative or stable replies from at least one correct replica i in view v such that h = *prefix(n, v, i)*. Since this replica has established v, there exists an established history h'' = *e-hist(v, i)* = *t-hist(v, i)* = *prefix(*n''*, v, i)*. From *complete(*o'*,* n'*,* v'*,* h'*,* c'*)*, $v' < v$ and Lemma 11, h' is a prefix of h''. It follows that h' = *prefix(n, v, i)* and thus that h, h' and h'' are all prefixes of i in view v. As $n \leq n'$, h is a prefix of h' from Lemma 2.

Theorem 5. Property 1 holds.

Proof: Two clients can complete requests by receiving enough replies from replicas in the same view. If they complete their operations in the same view, the result follows from Lemma 12. Else, it follows from Lemmas 13 and 14.

A.1.5 Scrooge Liveness

The liveness property of Scrooge is the following:

Property 2. *If a correct client issues a request, it eventually completes it.*

Additionally, Scrooge ensures the following property:

Property 3. *If the system is in a timely period, v is the current established view for all correct replicas, the primary of v is correct and faulty clients only crash, eventually all correct clients complete their requests from speculative replies.*

The proofs assume that the system eventually enters a *timely period* where no timeout is fired and all sent messages are received. First it is shown that Property 2 holds.

Lemma 15. If the system is in a timely period and there exists a view v' such that the primary of v' is correct and all correct replicas initiate a view change to v', then all correct replicas eventually establish v'.

The view change protocol can block under this hypothesis because the *recover* function never completes correctly or because a new view can not be established. It is now shown that the protocol does not block in either case.

For recovery, the new primary will eventually receive $N - f$ well-formed view change and check messages from correct replicas. These also eventually satisfy the predicate STABLE as the vector res of each check message in CH contain binary values (see Lines 2.20 – 2.22) and each correct replica eventually sends its own outcome vector for each of these view change messages. As the system is composed of at least $N - f \geq f + 2b > 2b$ correct replicas, at least one of the two outcomes collects $b + 1$ check messages. Therefore, for recovery to block, either the predicate WAIT-ORD or the predicate WAIT-AGREED must still hold after $N - f$ view change and check messages are received from correct replicas and $|VC| \geq N - f$ (Lines 3.20 – 3.21).

For WAIT-ORD, let $mv < v'$ be the highest view $vc.v$ reported by a view change message $vc \in VC$ (Lines 3.11 – 3.15). If the old primary of $p(mv)$ were correct, each history element stored by a correct replica in view mv would be consistent with those of the primary of that view. If $p(mv)$ were correct and only correct replicas would have sent view change messages to the new primary of view v, there would not be inconsistent candidates. As multiple inconsistent candidates are present and one of them is sent by $p(mv)$, it follows that at least one Byzantine replica, either a backup which reports a forged element of the old primary, has sent a view change message which is included in $|VC|$. This implies tat $|VC| > N - f$ so WAIT-ORD does not hold.

For WAIT-AGREED to hold, the primary of view v' must have received an agreed candidate e for sequence number n and view mv. According to the predicate definition, this implies that (i) at most b correct replicas have associated e in their local history for n and mv is their last established view, and (ii) at most $f+b$ correct replicas have a view $v'' \neq mv$ as last established view or do not associate e to their agreed history prefix for sequence number n and view mv. From (i) and from the fact that mv is the highest established view contained in any view change message received by the primary of the new view v, it follows that at least $f+b$ correct replicas have not yet established mv or have established mv but have not included e in their local history for n. Therefore, from Lemma 5, no correct replica in view mv can include e in their agreed history prefix for sequence number n. This implies that all $N-f$ correct replicas either have a view $v'' \neq mv$ as their last established view, or do not associate e to their agreed history prefix for sequence number n and view mv. This contradicts (ii). Therefore, WAIT-AGREED does not hold and the protocol does not block due to the recover function.

After recovery is concluded, the new correct primary sends new view messages to all correct replicas, which then compute the same decision procedure as the primary in Lines 2.24 – 2.28 and send consistent establish view messages. Lines 2.30 – 2.34 can thus be completed and the new view is established.

Lemma 16. If the system is in a timely period, v is the current view established by all correct replicas and a correct replica sends an agreement message for sequence number n and view v, then all other correct replicas eventually do the same.

Proof: If a correct replica executes Lines 14.9 – 14.11 for n, then every other replica will receive an agreement message for n. If the replica i has already received an order request for n, it starts agreement in Lines 13.23 – 13.28. Else, the agreement message makes the predicate AGREEMENT-STARTED(i, n, v) hold. Agreement is started by replica i when the replica receives the order request for sequence number n (Line 1.26).

Lemma 17. If the primary of a view v is correct, the system is in a timely period and v is the current view established by all correct replicas, then a view change is never initiated by a correct replica and all requests from correct clients are completed.

Proof: A correct replica which does not suspect the primary and never executes Lines 13.42 – 13.44 initiates view change only if at least another correct replica accuses the primary (Lines 2.15 – 2.16). A correct replica accuses the primary of the current view v in Lines 13.42 – 13.44 only if it starts an agreement phase for a sequence number n in Lines 14.9 – 14.11

and the timer expires. From Lemma 16, if a correct replica starts agreement each other correct replica do the same. If the primary is correct and the system is timely, the agreement phase is concluded by each correct replica (Lines 13.23 – 13.28). This implies that the commit phase is also concluded before the timer at any correct replica expires (Lines 13.30 – 13.40) and consistent stable replies are sent by all the $N - f$ correct replicas. A correct client can thus complete all its requests in Lines 13.13 – 13.14 by re-sending them to all replicas.

Theorem 6. Property 2 holds.

Proof: Consider the system behavior when the the system eventually enters a timely period. If the correct client cannot complete a request from speculative replies in view v in Lines 1.28 – 1.30, it contacts all correct replicas until it can deliver a reply in Lines 13.13 – 13.14. When correct backups receive from the client a request in Lines 13.16 – 13.21, they start a timer and accuse the primary if the reconfiguration phase is not completed when this expires (Lines 13.42 – 13.44). If the primary is correct, all correct replicas obtain an order request, start agreement in Lines 1.10 – 1.12 or 13.16 – 13.21 or 1.26, and complete the agreement and the commit phases. Since there are at least $N - f$ correct replicas in the system, the client can receive $b+1$ consistent stable replies and complete its requests in Lines 13.13 – 13.14. If the primary is faulty and less than $b + 1$ correct replicas conclude the commit phase and send a stable reply, at least $f + b$ replicas timeout and send a view change message to all other replicas. From Lines 2.15 – 2.16, all correct replicas also start a view change to remove the faulty primary. This is iterated until either the client completes the request in a view, or all replicas execute a view change to a view v' with a correct primary. From Lemma 15, the view change to v' is completed by all correct replicas. From Lemma 17, no correct replica initiate a further view change and all requests from correct clients are completed.

Scrooge also ensures the following additional liveness property to re-establish speculation after a failure event.

Theorem 7. Property 3 holds.

Proof: Consider the system behavior when the the system eventually enters a timely period. The first step is proving that SPEC-RUN eventually holds for each correct replica i. The primary proposes a replier quorum $RQ_p = Q$ along with an order request for sequence number n. If SPEC-RUN does not hold for a correct replica i, the replica send an agreement message for n in Lines 1.23 – 1.26. From Lemma 16, all other correct replicas do the same. If the primary never changes its replier quorum again, this implies that no client re-sends a request suspecting a replica in Q because all requests are completed from speculative replies, q.e.d. Let us thus assume by contradiction that n' is the lowest sequence number after n where the primary associate a replier quorum $RQ_p \neq Q$ to an ordered sequest. As the system is timely, all correct replicas commit on sequence number n and send speculative replies for all sequence numbers in (n, n'). Replicas do this in Lines 1.15 – 1.22 if they receive the order request for the sequence number after the commit is reached, or in Line 13.40 otherwise. The primary updates its suspect list in Lines 1.10 – 1.12 and proposes a different replier quorum $RQ_p = S$ for n' only if the client of a request with sequence number in (n, n') has suspected some replica in Q as faulty and has indicated this while re-sending the request (see Lines 14.1 – 14.6). The client re-sends a request only if it has not received a speculative reply from at least one replica in Q, which is thus faulty. If $f = 1$, the new replier quorum S does not contain the faulty replica. S is eventually committed as, as explained previously, all replicas send speculative replies for all sequence number greater than n', a contradiction. If $f > 1$, the result follows by simple induction on the number of faults detected by clients.

A.2 Integrating Garbage Collection

This subsection describes how garbage collection is integrated intro Scrooge. Readers who are familiar with PBFT [CL99] will notice that Scrooge uses the very same mechanisms.

A.2.1 Garbage Collection

In BFT replication protocols, the checkpoint subprotocol is used to curb the size of the message history. With checkpoiting, replicas only store history elements with sequence numbers in the range $[l + 1, l + L]$ where l is called lower watermark, L is the size of the history log, and $l + L$ is called higher watermark.

Given a constant checkpoint interval K, a tentative checkpoint of the application state is built after requests with sequence number n such that $(n \bmod K) = 0$ are executed. The checkpoint subprotocol indicates that a tentative checkpoint can be retrieved by any correct replica because it has been established by enough (i.e. $b + 1$) correct replicas. When this happens, the checkpoint is considered as *stable*, all history elements prior to n are garbage-collected, and the lower watermark l is set to n. The protocol steps are the following:

Step GC.1: Replica executes the n^{th} *request and* n *mod* $K = 0$
Replica i initiates an agreement phase by executing the procedure *agree* for the request.

Step GC.2: Replica commits the n^{th} *request and* n *mod* $K = 0$
Replica i builds a tentative checkpoint composed by the application state and the replier quorum associated with sequence number n in the history, and sends a checkpoint message $\langle$CHECKPOINT, n, $i\rangle_{\mu_i}$ to all other replicas.

Step GC.3: Replica receives a checkpoint message
If a replica i receives $f + b + 1$ checkpoint messages corresponding to one of its tentative checkpoints for sequence number n, it considers it as *stable*, it sets the lower watermark l to n and garbage-collects the history elements and tentative checkpoints for sequence numbers $n' < n$.

A.2.2 Modifications to Normal Executions

Replicas simply need to check that their history log does not overflow.

Lines 1.6 – 1.13: Primary receives a request
The primary assigns sequence number n to the request only if n is not greater than its higher watermark $l + L$.

Lines 1.15 – 1.26: Primary receives an order request
A replica accepts an order request message *or* only if its sequence number *or.n* is not greater than its higher watermark $l + L$.

A.2.3 Modifications to View Change

With checkpointing it is not necessary to recover the entire history of previous requests but only a small subset of it. Replicas include information about their current checkpoint in their view change messages. An initial checkpoint for the new view is selected by the recovery function based on this additional information. The specific modifications are the following:

Lines 2.1 – 2.5: Initiating view change
The format of the view change message is modified to $\langle$VIEW-CHANGE, v', v, mh, C, E $i\rangle_{\sigma_i}$, where C is a set containing, for each checkpoint stored by the replica, a tuple $\langle n, d, RQ\rangle$ where n is the sequence number where the checkpoint was taken, d is the digest of the application state, and RQ

is the repliers quorum associated to n. Also, the message history mh only includes the messages which are currently in the log and have not yet been garbage-collected.

Recover function: Recovering the observed history

The primary selects the checkpoint tuple $\langle n', d, RQ_{n'} \rangle$ with the highest sequence number n' which is contained in the view change messages of at least $b+1$ replicas and such that the view change messages of at least $f+b+1$ replicas report checkpoints for sequence numbers $n \leq n'$. This is called *initial checkpoint*. The history is then recovered as in the previous case but only for sequence numbers in the range $[n'+1, n'+L]$, where L is the size of the history log. As in the previous case, the last established view mv is still identified using view establishment certificates, but the entire initial history ih_v is not recovered. Therefore, the instructions after Line 3.11 and until 3.15 are removed. $RQ_{n'}$ is used to identify observed candidates for sequence number $n'+1$.

Lines 2.24 – 2.28: Backup receives a new view message

Backup replicas also perform the same steps as the primary to recover the initial history for the new view.

A.2.4 Correctness

In the following, it is proven that checkpointing preserves both safety and liveness. For safety, the following must be proven.

Lemma 18. If an initial checkpoint $\langle n', d, RQ_{n'} \rangle$ is selected and L is the size of the history log, then d and $RQ_{n'}$ are respectively the only digest of the application checkpoint and the only replier quorum associated with sequence number n' by any correct replica in any view.

Proof: A initial checkpoint is selected only if it has been sent by $b+1$ replicas, including a correct one. This correct replica has thus completed the commit phase for sequence number n'. It follows from an argument similar to those of Lemmas 8 and 11 that the agreed history prefix for n' is recovered in any view by any correct replica.

Lemma 19. If an initial checkpoint $\langle n', d, RQ_{n'} \rangle$ is selected and L is the size of the history log, then no request with sequence number greater than $n'+L$ has completed.

Proof: Let us assume by contradiction that a request r is completed with sequence number n greater than $n'+L$. If r is completed in Lines 1.28 – 1.30 or 13.13 – 13.14, at least $N-f$ replicas have accepted an order request message with sequence number n. From Lines 1.15 – 1.26 it follows that n is not greater than their higher watermark. This implies that the lower watermark of these $N - f$ replicas is strictly greater than n' and, from Step GC.3, that their checkpoint for n' has been garbage-collected. At most f correct replicas and b Byzantine replicas can thus report a checkpoint for n' in their view change messages. This checkpoint can not be chosen as initial checkpoint by the recovery function as it is included in the view change messages from at most $f + b$ replicas.

Liveness is also ensured as follows.

Lemma 20. A correct replica can always recover one provably correct checkpoint.

Proof: Consider a period where the system is timely and let $c = \langle n, d, RQ_n \rangle$ be the stable checkpoint with the highest sequence number among those established by any correct replica at any given moment t. The next step is proving that there are at least $b + 1$ correct replicas storing c as tentative or stable checkpoint c. This ensures that, by receiving $b+1$ consistent checkpoints from these replicas, any other correct replica can prove that the checkpoint is correct. Assume by contradiction that at most b correct replicas store c. This implies that a set Q of at least $f + b$ correct replicas only store checkpoints for either smaller or larger sequence numbers than n. As a correct replica has set c as stable checkpoint, at least $f + b + 1$ replicas have once stored c as tentative checkpoint (Step GC.3). It is thus impossible that all the $f + b$ correct replicas in Q only store checkpoints for sequence numbers smaller than n. At least one of them, say j, must have only stored checkpoints for sequence numbers larger than n. This implies that the tentative checkpoint c has been garbage-collected by j because a higher stable checkpoint has been reached. Therefore, c is not the stable checkpoint with the highest sequence number among those established by any correct replica at time t, a contradiction.

Algorithm 13: Scrooge - Explicit agreement

```
13.1  procedure agree($m$)
13.2      if $\exists k : mh[k].m = m$ and never sent agree message for sequence number $k$ in view $v$ then
13.3          send $\langle$AGREE, $v$, $k$, $h[k]$, $i\rangle_{\mu_i}$ to all replicas;
13.4          start timer if not already running;
13.5
13.6  upon client timeout
13.7      $SL \leftarrow \perp$;
13.8      if $\exists RQ$ : received matching speculative replies $sp$ to $m$ with $sp.RQ = RQ$ from a set $S \subset RQ$
          of $N - 2f$ replicas then
13.9          $SL \leftarrow RQ \setminus S$;
13.10     stop waiting for $sp$ messages; $timer \leftarrow timer \cdot 2$;
13.11     repeat
13.12         send $m = \langle$REQ, $o$, $t$, $c$, $SL\rangle_{\sigma_c}$ to all replicas;
13.13     until client receives $b + 1$ matching stable replies $st$ to $m$ ;
13.14     deliver ($o$, $t$, $st.r$);
13.15
13.16 upon backup $i$ receives request $m$ from client $m.c$
13.17     if not IN-HISTORY($m$, $mh$) then
13.18         send $m$ to primary $p(v)$;
13.19         start timer if not already running;
13.20     else if not COMMITTED($m$, $mh$, $cw$) then agree($m$);
13.21     else reply-cache($m.c$);
13.22
13.23 upon replica $i$ receives an agree message $ag$ from replica $ag.i$
13.24     if $ag.v = v$ and $ag.h = h[ag.n]$ then
13.25         agree($mh[ag.n].m$);
13.26         if received $N - f - 1$ matching agree messages for $ag.n$ from other replicas then
13.27             send $\langle$COMMIT, $v_i$, $n$, $i\rangle_{\mu_i}$ to all replicas;
13.28             $aw \leftarrow ag.n$;
13.29
13.30 upon replica $i$ receives a commit message $cm$ from replica $cm.i$
13.31     if $cm.v = v$ and $cm.n \leq aw$ and received $N - f - 1$ matching commit messages for $cm.n$
          from other replicas then
13.32         $c \leftarrow mh[cm.n].m.c$; $t \leftarrow mh[cm.n].m.t$;
13.33         $r \leftarrow$ stored reply for $mh[cm.n]$;
13.34         send $\langle$STAB-REP, $v$, $n'$, $c$, $t$, $r$, $i\rangle_{\mu_{i,c}}$ to client $c$;
13.35         if $cw \leq cm.n$ then
13.36             $cw \leftarrow cm.n$; $RQ_{cw} \leftarrow mh[cw].RQ$;
13.37             if $\forall k \in [cw, n] : mh[k].RQ = RQ_{cw}$ then
13.38                 $RQ \leftarrow RQ_{cw}$;
13.39         if never sent agree message for sequence number $n' > cw$ and view $v$ then stop timer;
13.40         send-missing-spec-rep($cw$, $RQ_{cw}$);
13.41
13.42 upon replica timer expires
13.43     $timer \leftarrow timer \cdot 2$;
13.44     view-change($v' + 1$) ;
13.45
```

Algorithm 14: Scrooge - Helper procedures

```
14.1  procedure update ($SL'$)
14.2      if $n > n_{SL}$ and $|SL'| \leq f$ then
14.3          $n_{SL} \leftarrow n$;
14.4          if $p(v) \in SL'$ then $SL' \leftarrow SL' \setminus \{p(v)\}$;
14.5          remove the $|SL'|$ oldest elements from $SL$;
14.6          add elements of $SL'$ into $SL$;
14.7
14.8  procedure agree($m$)
14.9      if $\exists k : mh[k].m.c = m.c$ and $mh[k].m.t = m.t$ and never sent agree message for sequence
          number $k$ in view $v$ then
14.10         send $\langle$AGREE, $v$, $k$, $h[k]$, $i\rangle_{\mu_i}$ to all replicas;
14.11         start timer if not already running;
14.12
14.13 procedure reply-cache($c$)
14.14     $n' \leftarrow$ sequence number of last committed operation from $c$;
14.15     $r \leftarrow$ stored reply for $mh[n']$;
14.16     send $\langle$STAB-REP, $v$, $n'$, $c$, $mh[n'].t$, $rc[n']$, $i\rangle_{\mu_{i,c}}$ to client $c$;
14.17
14.18 procedure send-missing-spec-rep($k$, $RQ$)
14.19     if $i \in RQ$ then
14.20         while $mh[k].RQ = RQ$ and never sent speculative reply for sequence number $k$ in view
              $v$ do
14.21             $m \leftarrow mh[k].m$;  $r \leftarrow$ stored reply for $mh[k]$;
14.22             send $\langle$SPEC-REP, $v$, $k$, $h[k]$, $RQ$ $m.c$, $m.t$, $r$, $i\rangle_{\mu_i}$ to client $m.c$;  $k \leftarrow k + 1$;
14.23
```

Appendix B

Correctness of HeterTrust

This section proves that HeterTrust satisfies the specified properties of a trustworthy replicated service. An *operation* is *pending* if it is invoked by a client but never completed. An operation is *proposed* if it is issued by a leader in a PROPOSE message with proposal number *prop*. A *proposal* is the pair $(op, prop)$. A proposal, and therefore the associated request and the corresponding reply, is *accepted* by a coordinator if this accepts the corresponding PROPOSE message. This happens if the proposal comes from a leader that the coordinator currently endorses, or a following one with a higher proposal number. As coordinators receive proposals through $f+1$ servers, Lemmas 21 and 22 guarantee that accepted proposals have been sent by a leader coordinator and replied by at least one correct server. If the operation is accepted by a coordinator, it is accepted together with the corresponding correct reply. A proposal is *chosen* for a sequence number k' if a majority of coordinators accepted it for k'. An operation is chosen for a sequence number k' if it is contained in a proposal accepted for k'. At most one operation can be chosen for each sequence number (Lemma 23). Based on this property, clients and servers can take irreversible actions on operations (i.e., deliver them and commit them) if they receive a majority of ACCEPTED messages and thus learn that the operation was chosen (Lemma 24). A request is *retrievable* if it is chosen for a sequence number i and $g+1$ coordinators have learnt it. As leaders continue sending requests for a sequence number until they become retrievable, eventual progress is guaranteed even if correct servers are temporarily disconnected and do not commit old requests (Lemma 25). Finally, the required properties of the protocol are proven by Theorem 8.

Lemma 21. *Only an operation that has been proposed by a leader coordinator is accepted for a given proposal, together with a reply obtained from at least one correct server.*

Proof. By definition, a request is accepted for a sequence number k' only if it is contained in a proposal $(op, prop)$ that is accepted by any coordinator (line 5.16). A coordinator accepts a proposal $(op, prop)$ for k' only after it receives $f + 1$ equal (EXECUTED,$op, k', prop, repl$) messages from different servers (line 5.15). Among these servers, at least one must be correct. This has thus sent the EXECUTED message containing values k', op and $prop$ as from the message proposed by a leader coordinator and the reply $repl$. □

Lemma 22. *Only an operation that has been proposed by a leader coordinator is chosen for a given pair of proposal and sequence numbers, together with a reply obtained from at least one correct server.*

Proof. This follows directly from Lemma 21 as a chosen request must be also accepted. □

Lemma 23. *At most one operation can be chosen for a sequence number k'.*

Proof. An operation op is chosen for k' when a proposal $(op, prop)$ is chosen, i.e., it is accepted by a majority of coordinators. By definition, only one proposal can be chosen at a time. Assume that $p_1 = (op_1, prop_1)$ is the first proposal chosen for k'. By Lemma 22, p_1 is proposed by a leader coordinator l_1. In order for any another proposal $p_2 = (op_2, prop_2)$ with $op_1 \neq op_2$ to be chosen, it is necessary that at least one of the coordinators that accepted p_1 accepts p_2 afterwards.

From Lemma 21, any accepted value has been proposed by a leader. As leaders never change their proposals until demoted and re-elected, p_2 must have been issued with proposal number $prop_2 \neq prop_1$. Therefore, a coordinator accepts the new proposal p_2 after having accepted $p_1 = accval[k']$ only if p_2 has a higher proposal number $prop_2 > prop_1$ (line 5.13). The next step is showing that any chosen proposal p_2 issued after p_1 is such that $op_2 = op_1$.

The proof is by contradiction. Let us assume that p_2 is the proposal with the minimum proposal number $prop_2 > prop_1$ such that $op_2 \neq op_1$. The general case when p_2 is such that $prop_2 > prop_1$ follows by simple induction.

When l_2 is elected, it sends a QUERY message to all coordinators and sends new proposals only after it receives ENDORSE messages from a majority of them (lines 7.1–7.10). At least one of the coordinators member of the majority which accepted $p_1 = accval[k']$ must have sent an ENDORSE message reporting either that (a) op_1 is retrievable ($i \in Retr_{co}$) or (b) p_1 was accepted ($p_1 \in Acc_{co}$) (lines 7.27–7.30). In the first case l_2 does not send any new proposal for i (line 7.16). Therefore, if l_2 proposes p_2 for k' instead of p_1, there must exist a coordinator co reporting in an ENDORSE message to

l_2 that it has accepted a proposal p_3 from a leader l_3 with proposal number $prop_3 > prop_1$ (lines 7.16–7.19). Note that $prop_3 \neq prop_2$, since l_2 has not yet started making proposals with proposal number $prop_2$ at this point. This implies that *co* has set $maxProp = prop_3$ when the proposal was accepted (line 5.14). Since *co* replies to the QUERY message of l_2 after accepting p_3, it holds $prop_2 > prop_3$ (line 7.25). Therefore, p_2 is not the accepted proposal with the minimum proposal number greater than $prop_1$, a contradiction. □

Lemma 24. *Only a reply to a chosen operation can be delivered by a client, and only a chosen operation can be learnt by a coordinator or committed by a server.*

Proof. Coordinators send an ACCEPTED message containing a proposal only after accepting it (lines 5.16–5.18). Receiving ACCEPTED messages from a majority of coordinators is a necessary condition for clients to deliver a reply (line 4.7). Coordinators and servers learn that a request is chosen either by the same condition (lines 5.21 and 6.15), or by receiving a LEARNT message (line 5.25), which is sent only after some coordinator has learnt that the request was chosen (line 5.23). A server commits an operation only after it learns that it is chosen, and executes it unless it has already been tentatively executed (lines 6.15 and 6.22–6.26). □

Lemma 25. *For every sequence number k', eventually either there exist no pending operations or some operation is chosen for k' and becomes retrievable.*

Proof. The proof is by induction on the sequence numbers, assuming that a *no_op* request with sequence number 0 is trivially chosen and retrievable.

Assume that requests for all sequence numbers $k'' < k'$ have been chosen and are retrievable. Assume by contradiction that some client has an operation *op* that remains pending forever but no operation is chosen and becomes retrievable for k'. By repeatedly sending its request (line 4.11–4.13), the client can ensure that each request is eventually received by all coordinators.

By the property of Ω, eventually a single correct leader is elected. Eventually, this leader sends QUERY messages with a proposal number that is accepted by a majority of coordinators, and completes recovery. If a proposal for k' is not retrievable then the leader proposes some operation for k', either *op* or some other operation, until the operation becomes retrievable for k' (lines 5.35–5.38). As all requests with sequence numbers $k'' < k'$ are retrievable, correct servers can eventually obtain them from at least one correct coordinator (lines 6.34–6.37 and 5.31–5.33) and commit them (line 6.21). Eventually $s - f \geq f + 1$ correct servers can process the operation for sequence number k' (lines 6.4–6.10) and send the corresponding EXECUTED message to the coordinators, which then accept the proposal (line 5.16). The $c - g \geq \lceil (c+1)/2 \rceil$ correct coordinators forward ACCEPT messages to each other, until eventually all of them will learn that the operation is chosen (line 5.21–5.22) and make it eventually retrievable, by exchanging LEARNT messages (lines 5.23, 5.25 and 5.33).

□

Theorem 8. *The HeterTrust protocol satisfies the properties of Termination, Uniform Agreed Order, Update Integrity and Response Integrity.*

Proof. Termination: Assume by contradiction that some client operation remains pending forever. From Lemma 25, some proposed operation is eventually chosen for each sequence number, and becomes retrievable. In a fair run, eventually also the client operation will be chosen and become retrievable for a sequence number. The client resends the same request until receives ACCEPT messages from a majority of coordinators (since the operation is chosen) and delivers a reply, a contradiction.

Uniform Agreed Order: A correct server commits only chosen operations (Lemma 24). If the operation *op* is committed, and thus chosen, with sequence number k', Lemma 23 ensures that any other correct server that commits an operation with sequence number k' will commit *op*.

Update Integrity: If a server commits an operation *op* for a sequence number k, then every future execution of *op* will return a cached value (line 6.9). Therefore, a committed operation is never executed again. Furthermore, each committed operation $op \neq no\_op$ is issued by a client. In fact, only chosen operations are committed (Lemma 24), only proposed operations are chosen (Lemma 22) and an operation *op* is proposed by a leader coordinator only if it is received from a client (line 5.1).

Response Integrity: As coordinators are physically interposed between servers and clients, clients can receive replies *rep* (as well as any other data) from servers only through ACCEPTED messages sent by coordinators. These are sent only for accepted operations, which are associated with replies obtained from at least one correct server and sent by a correct coordinator upon receiving a client request (Lemma 21).

□

Appendix C

Eventual Linearizability

This Appendix first shows the locality and nonblocking properties of Eventual Linearizability (Appendix C.1). It then shows that Eventual Consensus is necessary and sufficient to implement of Eventual Linearizability, while Eventual Consistency is not sufficient (Appendix C.2). Finally, it shows the correctness of the Aurora protocol (Appendix C.3).

C.1 Locality and nonblocking

This section shows that Eventual Linearizability inherits the most relevant properties of Linearizability as it is both *local* and *nonblocking*. Locality ensures that if every object of a system is eventually linearizable, then the system itself is also eventually linearizable. Being nonblocking implies that the specification of Eventual Linearizability does not result in runs where some process can not make progress any longer.

In order to define locality, an additional definition is needed. An *object subhistory* $H|x$ of an object x is the history composed by all events in H referring to x. A history H is (t, L)-linearizable if L is a t-linearization of H.

The following two lemmas prove that weakly consistency and t-linearizability are local properties, which imply the locality of Eventual Linearizability.

Lemma 1. *If a history H is weakly consistent then, for each object x, $H|x$ is weakly consistent. If $H|x$ is weakly consistent for each object x, then H is weakly consistent.*

Proof. Since H is weakly consistent, then for every process p_i and operation o completed by p_i in H there exists a legal sequential history $\tau(i, o)$ which fulfills (i)-(iii). If o is an operation of x, then $H|x$ and $\tau(i, o)|x$ also fulfill (i)-(iii). Otherwise, o is not invoked in $H|x$. Therefore, $H|x$ is also weakly consistent.

On the other hand, given that $H|x$ is weakly consistent and $\tau(i,o)$ fulfills (i)-(iii) for every process p_i and operation o completed by p_i in $H|x$, o is also completed in H by the same process and $\tau(i,o)$ is legal sequential history of H too. Therefore, H is also weakly consistent. □

Lemma 2. *If a history H is t-linearizable then, for each object x, $H|x$ is t-linearizable. If $H|x$ is t_x-linearizable for each object x, then H is t_{max}-linearizable with $t_{max} = \max_{\forall x}(t_x)$.*

Proof. It is evident from the definitions that if H is t-linearizable then $H|x$ is t-linearizable for each object x. In fact, if L is a t-linearization of H, then $L|x$ is a t-linearization of $H|x$ and all response events in $L|x$ after t have the same results as in $H|x$. Therefore, $H|x$ is $(t, L|x)$-linearizable for each object x.

In order to prove the second implication, assume that for each x, $H|x$ is t_x-linearizable. Let R_x be the response events added to $H|x$ to build the t_x-linearization L_x of $H|x$, and H' the history obtained from appending all events of R_x to H. Let $<_x$ be the total order of all operations in $H|x$ defined by L_x, and $<$ be a relation built as the transitive closure of $\bigcup_{\forall x} <_x \cup <_{H,t_{max}}$. Assuming that $<$ is a partial order, it is possible to build a t_{max}-linearization L of H which respects $<$. For each x, all operations on x are ordered in L as in L_x. This implies that the results of the response events in L are the same as in $L|x$. Since $H|x$ is $(t_x, L|x)$-linearizable, all response events of H after $t_x \leq t_{max}$ have the same results as in L, so H is (t_{max}, L)-linearizable and thus t_{max}-linearizable.

The next step is showing that $<$ is a partial order. Assume by contradiction that $o_1 < \ldots < o_n$ and $o_n < o_1$, where $<$ can be either $<_x$ for some x or $<_{H,t_{max}}$, and assume that this is a cycle with minimal length in $<$. If all these operations are on the same object x, then they are totally ordered by $<_x$. The existence of a cycle implies that there must exist two operations o_i and o_j on x such that $o_i <_x o_j$ and $o_j <_{H,t_{max}} o_i$. This contradicts with (P2) as $<_x$ is the order of a t_x-linearization L_x of $H|x$ and (P2) implies that $<_{H,t_x} \subseteq <_x$. This and $<_{H,t_{max}} \subseteq <_{H,t_x}$ imply that $<_{H,t_{max}} \subseteq <_x$, a contradiction.

The cycle must thus contain operations on at least two objects. Assume o_i is an operation on object x. Let o_k be an operation in the cycle on a different object than x and such that $o_{(k+1 \bmod n)}, \ldots, o_{(i-1 \bmod n)}$ are on x. Similarly, let o_j be an operation in the cycle on a different object than x and such that $o_{(i+1 \bmod n)}, \ldots, o_{(j-1 \bmod n)}$ are on x. Since $o_k < o_i < o_j$, it follows that $o_k <_{H,t_{max}} o_i <_{H,t_{max}} o_j$, so $o_k <_{H,t_{max}} o_j$. It must thus hold $k \neq j$, which implies that a cycle exists $o_1 < \ldots < o_k < o_j < \ldots < o_n$ that is shorter than the one with minimal length, a contradiction. $\square$

It is now shown that Eventual Linearizability is nonblocking by showing that weakly consistency and t-linearizability are nonblocking.

Lemma 3. *Let inv be an invocation of a total operation o on an object x. If inv on x is invoked by a process p_i in a weakly consistent history H, then there exists a response resp on x of p such that the history H' obtained by appending resp to H is weakly consistent.*

Proof. Given an operation o' completed by process p_j in the weakly consistent history H (resp. H'), the corresponding legal sequential history is denoted by $\tau_H(j, o')$ (resp. $\tau_{H'}(j, o')$). Let operation o' be the last completed operation in H invoked by process p_j. Then, *resp* is determined by the execution $\tau_H(j, o') \cdot inv \cdot resp$. In H', o is a completed operation. Let $\tau_{H'}(i, o)$ be $\tau_H(j, o') \cdot inv \cdot resp$. For every other operation o' completed by p_j in H', $\tau_{H'}(j, o')$ equals $\tau_H(j, o')$. As a result, $\tau_{H'}(i, o)$ and every $\tau_{H'}(j, o')$ fulfill (i)-(iii) because H is weakly consistent and $<_H = <_{H'}$. $\square$

Lemma 4. *Let inv be an invocation of a total operation on an object x. If inv on x is invoked by a process p in a t-linearizable history H, then there exists a response resp on x of p such that the history H' obtained by appending resp to H is t-linearizable.*

Proof. Let L be a t-linearization of H. If L includes a response to *inv*, q.e.d. If not, *inv* is not included in L since L only contains completed operations. Since the operation is total, there exists a result for a response event *resp* that is determined by the execution of $L' = L \cdot inv \cdot resp$. L' is a t-linearization of H'. As *resp* has the same response in H' and L', H' is t-linearizable for any value of t such that H is t-linearizable. $\square$

Theorem 1. *Eventual Linearizability is nonblocking and satisfies locality.*

Proof. Directly follows from Lemmas 1, 2, 3 and 4. $\square$

C.2 Eventual Consistency, Eventual Consensus and Consensus

This section distinguishes between *high-level events*, which are executed on the interface between the application and the execution layer, and *low-level* *events*, which are executed on the interface between the execution layer and the consistency layer. Given a run σ of the system, $top(\sigma)$ is defined as the history containing all high-level events and $bot(\sigma)$ the history containing all low-level events. Eventual Linearizability constraints the set of admissible high-level histories $top(\sigma)$ of a run σ. The specifications discussed in this Section constraint the low-level histories $bot(\sigma)$ of a run σ.

Eventual Consistency is not sufficient to implement Eventual Linearizability for arbitrary objects. In fact, the following Theorem 2 shows that it is not even sufficient to implement an eventually linearizable register.

Theorem 2. *An eventually linearizable implementation of a single-writer, single-reader binary register cannot be simulated using only an eventually consistent consistency layer.*

Proof. Consider a system with two processes p_0 and p_1, where p_0 is a writer and p_1 is a reader. The register stores an initial value 0. Assume by contradiction that there exists an implementation of a read/write register with Eventual Linearizability using only an eventually consistent consistency layer. Let t_l be the time such that, for all runs σ such that $bot(\sigma)$ satisfies Eventual Consistency, t_l-linearizability holds for $H = top(\sigma)$. The contradiction is shown by using three finite runs. The last of these runs leads to a violation of t_l-linearizability.

In the first run σ_0, process p_0 writes the value 1 onto the register after time t_l. Let t_w be the time when the write operation completes and t_0 be the time when the last event of σ_0 occurs. Process p_1 takes no actions in this run. Let $bot(\sigma_0)$ satisfy Eventual Consistency in this run.

In the second run σ_1, process p_1 invokes a read operation after time t_w. Let t_r be the time when the read operation completes and t_1 be the time when the last event of σ_1 occurs. Process p_0 takes no action in this run. Since no write operation is invoked in this run, the read must return the initial value 0. Let $bot(\sigma_1)$ satisfy Eventual Consistency in this run too.

In the third run σ_2, p_0 and p_1 observe the same events until t_r as in σ_0 and σ_1 respectively. For indistinguishability, the read operation of p_1 returns 0 even if it is preceded by a write operation writing 1. At a time $t_2 > \max(t_0, t_1)$, the consistency layer delivers at both processes the same sequence S including all the operations submitted before t_2. These delivery events are the last events of σ_2.

In every t_l-permutation L of $H = top(\sigma_2)$, the write operation precedes the read so the read operation returns 1. This contradicts t_l-linearizability of H since the write and read operations are invoked after t_l but the result of the read in H is 0. Therefore, $bot(\sigma_2)$ must violate Eventual Consistency. It is now shown that it is not the case, which leads us to the final contradiction.

It is easy to see that if nontriviality, set-stability and liveness hold for $bot(\sigma_0)$ and $bot(\sigma_1)$, then they also hold for $bot(\sigma_2)$. Prefix consistency holds if P_t is defined as follows. For $t \leq t_2$, P_t is equal to the empty sequence. For $t > t_2$, P_t is equal to the sequence S delivered at time t_2. This definition of P_t satisfies all properties (C1)-(C3) of prefix consistency. □

A consistency layer that satisfies Eventual Consensus satisfies *t-stability* if t is the time defined in the definition of Eventual Stability. Combining Eventual Consistency with Eventual Stability implicitly strengthens consistency. Namely, t-stability ensures *t-consistency*, which is defined as follows. A consistency layer satisfies t-consistency if for any correct processes p_i and p_j delivering at any times $t_i, t_j > t$, one of the sequences $S(i, t_i)$ and $S(j, t_j)$ is prefix of the other.

Eventual Consensus satisfies t-consistency.

Lemma 5. *If a consistency layer satisfies t-stability then it satisfies t-consistency.*

Proof. Assume that a consistency layer satisfies Eventual Consistency and eventual stability but contradicts the Lemma. Let t be the time after which stability holds. This implies that two delivery events occur at two processes p_i and p_j at times $t_i, t_j > t$ such that $S(i, t_i)$ and $S(j, t_j)$, which are the two sequences delivered at times t_i and t_j, are not prefix of each other.

If $i = j$ a contradiction follows directly eventual stability. Consider the case $i \neq j$. There must exists an index k and two different operations o_i and o_j that are the k-th elements of $S(i, t_i)$ and $S(j, t_j)$ respectively. It follows from eventual stability that for each $t'_i > t_i$ and $t'_j > t_j$, o_i and o_j that are the k-th elements of $S(i, t'_i)$ and $S(j, t'_j)$ respectively. From property (C3) of prefix consistency, there exists a time $tc_i > t_i$ such that P_{tc_i} includes o_i. From property (C1), P_{tc_i} must be a prefix of all $S(i, tc'_i)$ with $tc'_i > tc_i$ so o_i is the k-th element of P_{tc_i}. Similarly, from property (C3) and (C1) it follows that there exists a time tc_j such that P_{tc_j} includes o_j as the k-th element. However, P_{tc_i} and P_{tc_j} are not prefixes of each other. This violates (C2). □

Lemma 6. *An eventually linearizable implementation of an arbitrary object can be implemented using only a consistency layer satisfying Eventual Consensus.*

Proof. Assume that the consistency layer satisfies t_s-stability. From Lemma 5, the Eventual Consistency layer also satisfies t_s-consistency. The algorithm for the implementation is the one of Algorithm 8. High-level invocation events at each process p_i for each operation o are forwarded to the lower consistency layer. The implementation then waits for the first sequence delivered by the consistency layer at process p_i containing o. The time when this delivery event takes place is denoted as $t(o)$. The implementation then executes the resulting sequence and returns the results as an upper-layer response event.

It is clear from the liveness of the consistency layer and from Algorithm 8 that each invoked operation is eventually completed. Weak consistency directly derives from the set stability and nontriviality properties of the consistency layer. t_l-linearizability for some time t_l also holds, as it is now shown.

From the prefix consistency (C3) property of the consistency layer, all operations submitted by correct processes are eventually included in a consistent prefix P_t. From prefix consistency (C2), consistent prefixes are prefixes of each other. Let t_p be the minimum time such that all operations o such that $t(o) \leq t_s$ are included in P_{t_p}, and t_c be the minimum time when all faulty processes have crashed. Let t_l be the minimum time greater than $\max(t_s, t_p, t_c)$ and show that the simulation of Algorithm 8 satisfies t_l-linearizability.

Assume by contradiction that t_l-linearizability is violated. Since $t_l > t_c$ this implies that there exists, for some run σ, a high-level operation o_i of a correct process p_i in $H = top(\sigma)$ which is invoked after t_l and whose result is different than the result of o_i in any t_l-linearization L of H. Assume that there exists a t_l-linearization L of H having $S_i = S(i, t(o_i))$ as a prefix. It follows from the implementation of Algorithm 8 that the result of o_i in L is the same as in H, a contradiction. Therefore, there exists no such L. This implies that for some operation $o_k \in S_i$ there exists an operation o_j such that $o_j <_{H,t_l} o_k$ and $o_j \not<_{S_i} o_k$. This in turns implies that either $o_j \notin S_i$ or $o_k <_{S_i} o_j$. Before contradicting these two cases, note that from $o_j <_{H,t_l} o_k$, the completion of o_j precedes the invocation of o_k. From nontriviality, $S_j = S(j, t(o_j))$ cannot include o_k,

Assume that the first condition holds and that there exists an operation o_j invoked by a process p_j such that $o_j <_{H,t_l} o_k$, $o_k \in S_i$ and $o_j \notin S_i$. Consider two cases based on the value of $t(o_j)$. If $t(o_j) \leq t_s$ then o_j is included in P_{t_p}. From prefix consistency (C1), P_{t_p} is a prefix of S_i so $o_j \in S_i$. Therefore, $t(o_j) > t_s$ so it follows from t_s-consistency that one of S_i and $S_j = S(j, t(o_j))$ is a prefix of the other. Since $o_j \notin S_i$ but $o_j \in S_j$, S_i is a prefix of S_j, so $o_k \in S_j$. However, it has been already shown that $o_k \notin S_j$.

Consider now the second condition, that is, that there exist two operations o_j and o_k in S_i such that $o_j <_{H,t_l} o_k$ and $o_k <_{S_i} o_j$. Consider two cases. If $t(o_j) > t_s$, it follows from t_s-consistency that one of S_j and S_i are a prefix of each other. Since $o_k <_{S_i} o_j$ and $o_j \in S_j$, $o_k <_{S_j} o_j$. However, it has already been shown that $o_k \notin S_j$. If $t(o_j) \leq t_s$ then P_{t_p} includes o_j. From nontriviality and since o_k is invoked after $t_l \geq t_p$, there exists no process h such that $S(h, t_p)$ includes o_k. This and prefix consistency (C1) imply that P_{t_p}

does not include o_k. Since $t(o_k) > t_l \geq t_p$, P_{t_p} is a prefix of $S_k = S(k, t(o_k))$ from prefix consistency (C1). This implies that $o_j <_{S_k} o_k$. However, one of S_i and S_k is prefix of the other, a contradiction with $o_k <_{S_i} o_j$. In fact, from the definition of $<_{H,t_l}$, o_k is invoked after t_l. This and t_s-consistency imply that one of S_k and S_i are prefix of the other. □

Lemma 7. *A consistency layer satisfying Eventual Consensus can be implemented using only an eventually linearizable arbitrary object implementation.*

Proof. The proof shows that the simulation of Algorithm 9 satisfies Eventual Consistency and t_s-stability for some time t_s. Let t_l be the time such that t_l-linearizability holds for all histories, t_d be the time when all operations submitted by invoked processes before t_l are completed at all correct processes, and t_s be the minimum time greater than $\max(t_l, t_d)$. The existence of t_d is given by the liveness of the sequence implementation.

Set stability and nontriviality directly follow from the weak consistency property of the sequence. For liveness, it follows from the termination property of the sequence implementation that all append operations o invoked by a correct process upon a *submit(o)* event terminate. From t_l-linearizability, all appended operations are read by the first read operation o' invoked by each correct process after $\max(o, t_l)$. All submitted operations are thus appended and eventually delivered by each correct process.

For prefix consistency, let L be the t_l-linearization of the operations on the shared object, and let P_t be defined as follows. P_t is the empty sequence for $t \leq t_s$. For $t > t_s$, P_t is the value returned by the last read operation of any correct process which is ordered in L before all reads invoked by correct processes and ongoing at time t. Prefix consistency (C1) follows from the fact that every operation returned by a read invoked after t_d is observed by any following read in a t_l-linearization. For prefix consistency (C2) it is sufficient to observe that for each $t \geq t_l$ and $t' > t$, either $P_t = P_{t'}$ or the read whose return value defines $P_{t'}$ observes a sequence which is an extension of the sequence observe by the read of P_t. In fact, both sequences are prefixes of L. Prefix consistency (C3) directly follows from the liveness of the sequence implementation, from the definition of P_t and from the fact that sequences are periodically delivered. □

Theorem 3. *Eventual Consensus is a necessary and sufficient property of a consistency layer to implement arbitrary shared objects respecting Eventual Linearizability.*

Proof. The sufficiency of Eventual Consensus is shown by Lemma 6, the necessity is shown by Lemma 7. □

C.3 Correctness of the Aurora protocol

The proof starts by providing some additional definitions, notations and conventions which will be used in the following correctness argument.

C.3.1 Definitions

Time Some proofs refer to a global time reference $t \geq 0$. Computation time is ignored, and the state of a process at time t is the one after any event occurred at t. No two events occur at the same process and at the same time, and only a finite number of events occur in a finite time. A message is *received* or *abdelivered* when the corresponding receipt or abdelivery event occurs.

Sequences and histories Two sequence of operations are *compatible* if one of the two is a prefix of the other. A *strong prefix* is a prefix of operations terminating in a strong operations. Abusing the terminology, a sequence S_1 is a *subset* of another sequence S_2 if all operations of S_1 are included in S_2.

$H(i,t)$ is defined as follows: if p_i not crashed at time t, then $H(i,t)$ is the local history stored by p_i at time t; else, it is the last history stored by p_i before crashing. The order induced on operations by the local history of a process p_i at time t is denoted as $<_{i,t}$. The order on operations determined by a sequence S is denoted as $<_S$. Local variables and predicates of a process p_i are denoted by a subscript i.

A process p_i *stores* a variable $x = val$ *upon receiving* or *abdelivering* a message m if $x = val$ in the local state of p_i at the time of the local receipt or abdeliver event of m. A process p_i *stores* an operation at a given time if it includes the operation in its local history H_i. A process p_i *stores a strong prefix* π for round k when p_i stores a local history H_i containing the last operation of π upon abdelivering a PROP$(*,*,k)$, a PUSH$(*,k)$ or a CLOSE-RND(k) message. A process p_i *directly* stores a strong prefix π for round k when p_i stores π for k for the first time upon abdelivering a PROP$(*,*,k)$ or a CLOSE-RND(k) message. Strong prefixes that are indirectly stored by p_i are stored when p_i receives a PUSH message from some other process. π is a *longest strong prefix* for p_i at a given time t if π is a strong prefix of the local history $H(i,t)$ and there exists no strong prefix π' of $H(i,t)$ which is longer than π.

Sometimes it is necessary to show that the system converges to a common state after a certain time. Given a process p_i and a finite set of operations O, $t_w(i,O)$ is defined as the maximum time t in a given run when the following holds: at time t process p_i appends an operation $op = H'$ onto its history or executes a merge$(H', *, H, *)$ such that either (i) there exists $o \in O \cap H'$ such that $o \notin H(i,t)$, or (ii) there exist $o, o' \in H(i,t) \cap H'$ such that $o <_H o'$ and $o' <_{H'} o$.

Communication primitives The reliable channel module has the following property: if a correct process p_i sends a message m to a correct process p_j, then p_j eventually receives m. The atomic broadcast module has four properties: *(validity)* If a correct process abcasts a message m, then it eventually abdelivers m; *(uniform agreement)* If a process abdelivers a message m, then all correct processes eventually abdeliver m; *(uniform integrity)* For any message m, every process abdelivers m at most once, and only if m was previously abcast by its sender; *(total order)* If two correct processes p_i and p_j abdeliver two messages m and m', then p_i abdelivers m before m' if and only if p_j abdelivers m before m'.

Failure detectors and the quorum property The algorithm uses a failure detector $\mathcal{D}$ and a leader oracle $\Omega_{\mathcal{D}}$ implemented on top of $\mathcal{D}$. A *leader oracle* is any failure detector which outputs the id of a single *trusted* process. A correct process p_{ld} is *perpetually trusted at time* t if at each time $t' \geq t$ and for each correct process p_i, $\Omega_{\mathcal{D}} = ld$ at p_i. A correct process p_{ld} is *perpetually trusted* if it is perpetually trusted at some time.

A leader oracle is in class Ω_Q is it satisfies the following *quorum property*: there exists a quorum Q of correct processes and a process p_{ld} such that eventually all processes in Q perpetually trust p_{ld} and $|Q| > n/2$. Clearly, a leader oracle can satisfy this property only if a majority of correct processes exists. If this precondition is met, each leader oracle in Ω is trivially in Ω_Q. Furthermore, the simple Lemma 8 shows that given a leader oracle in Ω_Q, a leader oracle in Ω can be simulated using Algorithm 15, which relies on reliable FIFO channels. Therefore, classes Ω_Q and Ω are equivalent if a majority of correct processes exists.

Causal consistency Causal consistency is defined as follows. First let the *happens-before* relation $<_C$ be as follows. Let o and o' be two different operations, let i the the process that invoked o', and let t the time when o' is invoked. $o <_C o'$ if and only if $o \in S(i,t')$ for some $t' < t$ or there exists a third different operation o'' such that $o <_C o'' <_C o'$. A consistency layer satisfies *causal consistency* if, for each process i and time t it holds that: (C1) If $o \in S(i,t)$ and $o' <_C o$ then $o' \in S(i,t)$, and (C2) If $o, o' \in S(i,t)$ and o precedes o' in $S(i,t)$ then $o <_C o'$. It can be shown that this definition of causal consistency property is sufficient to implement causal memory [**?**].

Algorithm 15: Implementing Ω on top of $\mathcal{L} \in \Omega_Q$

```
Initially: $ld \leftarrow \perp$;
$T[j] \leftarrow \perp$ for each $j \in [0, n-1]$;
$\mathcal{L}$ outputs $\perp$;

// A process calls this function to query its local
   instance of the leader oracle
function query()
    if $ld \neq \perp$ then
        return $ld$;
    else
        $k \leftarrow \mathcal{L}$;
        return $k$;

upon $\mathcal{L}$ changes its output to $k$
    send TRUST_FD($k$) to all processes;

upon receive TRUST_FD($k$) from process $p_j$
    $T[j] \leftarrow k$;
    if $\exists h, Q : T[l] = h$ for each $l \in Q$ and $|Q| \geq n/2$ then
        $ld \leftarrow h$;
    else
        $ld \leftarrow \perp$;
```

C.3.2 Correctness proof

Lemma 8. *Algorithm 15 simulates a leader oracle in Ω using a leader oracle $\mathcal{L} \in \Omega_Q$.*

Proof. The proof is by contradiction. Assume that eventually the leader oracle $\mathcal{L}$ in Ω_Q permanently outputs the same process id k at a quorum Q of correct processes such that $|Q| > n/2$ and that p_k is not permanently trusted by the local instance of the simulation of some correct process. Each process in Q will eventually send a TRUST_FD(k) to all other processes as last TRUST_FD message. Since the communication channel is FIFO and reliable, these messages are eventually received by each correct process and are the last messages received from any process in Q. This implies that for each correct process, eventually it permanently holds $T[j] = k$ for each $j \in Q$. For each correct process p_i, when the last TRUST_FD message from a process in Q is received by p_i, ld is permanently set to k. The simulation thus permanently returns the same process id k to each correct process, a contradiction. □

Lemma 9. *If a process p_i abcasts a PROP(H', S, k) message m, then H' is an extension of a strong prefix π_{k-1} stored by p_i for round $k-1$ and $S \setminus H'$ is empty.*

Proof. Assume by contradiction that the thesis does not hold. It follows from the predicate *must-propose-new-prefix* that if p_i abcasts the PROP(H', S, k) message then H' is the local history of p_i, $S \setminus H'$ is empty and $k_i = k$. If p_i has already stored a strong prefix π_{k-1} for round $k-1$, H' is an extension of π_{k-1}, a contradiction. So p_i has not yet stored a strong prefix π_{k-1}. If p_i has set its local variable k_i to k then it has abdelivered a CLOSE-RND($k-1$) message when $P_i = (*, *, k-1, *)$. If $d_i < k-1$ upon abdelivering CLOSE-RND($k-1$), then p_i stored a strong prefix for π_{k-1} by doing the following merge, a contradiction. Therefore, $P = (*, *, k-1, *)$ and $d_i >= k-1$. This implies that p_i has already stored a strong prefix for $k-1$ upon abdelivering a PROP($*, *, k-1$) message or upon receiving a PUSH($*, k-1$) message, the final contradiction. □

Lemma 10. *If a process p_i directly stores a strong prefix for round k then p_i has stored exactly one strong prefix for round $k-1$ and $d = k-1$ when the strong prefix is stored for round k.*

Proof. The first step of this proof is showing that if p_i directly stores a strong prefix for round k at a certain time t_k, then it stores $k_i = k$ and $d_i < k$ immediately before t_k. Two events can induce p_i to directly store a strong prefix. If p_i stores a strong prefix upon abdelivering a PROP($*, *, k$) message, then from the definition of *proposal-stable* it must hold $k_i = k$ and $d_i <= k-1$, q.e.d. If the strong prefix is stored upon abdelivering a CLOSE-RND(k) message m, it must hold $P_i = (*, *, k, *)$ and, from the merge, $d_i < k$. From the definition of *from-round-winner*, P_i was assigned this value only if a PROP($*, *, k$) message m' is abdelivered before m and thus if $k_i = k > d_i$ at that time. It is now only needed to show that the values of k_i and d_i are not modified between receiving m and m'. This is easy to see for k_i. By contradiction, the value of d_i would be set to a value higher than $k-1$ before storing the strong prefix only if a PUSH($*, d$) message with $d > k-1$ is received. In this case p_i would not directly store a strong prefix for round k, a contradiction.

The next step is showing that at least one strong prefix has been stored by p_i for round $k-1$ and that $d_i \geq k-1$ immediately before t_k. The value of

k_i is set to k only upon abdelivering a CLOSE-RND$(k-1)$ message. When this occurs, a new strong prefix for round $k-1$ is included in the new history H_{new} built by p_i. If this strong prefix is stored by p_i in the subsequent merge, the proof is finished since d_i is set to $k-1$ by the merge and it holds $d_i \geq k-1$ until t_k since d_i monotonically grows. Else, this implies that p_i has already set $d_i = k-1$. This happens only if p_i has abdelivered PROP$(*, *, k-1)$ message and has stored a new strong prefix for $k-1$, or if it has received a PUSH$(*, d)$ message with $d = k-1$. In both cases process p_i stores a strong prefix for round $k-1$ and sets $d_i = k-1$, q.e.d.

It is only remains to show that no other strong prefix is stored for round $k-1$. This follows from the fact that $d_i \geq k-1$ after storing the first prefix for $k-1$ and that d_i monotonically grows. In fact, no following PROP$(*, *, k-1)$ message will lead p_i to the delivery of a strong prefix nor will any merge executed upon receiving a PUSH$(*, k-1)$ or a CLOSE-RND$(k-1)$ message do it. □.

Lemma 11. *The relation $<_{i,t}$ is a partial order for each process p_i and time t.*

Proof. Transitivity and reflexivity are trivial because histories are sequences. It is now shown that the relation is antisymmetrical, that is, it never induces cycles. Since a history is a sequence, it is sufficient to show that no local history has duplicates. This is trivially true for the initial empty history.

Histories are modified either by appending operations or by merging other histories. Assume by contradiction that an append or a merge creates a duplicate on a history for the first time. Appends of weak operations are always preceded by a check that an operation is not already present in the history. Appends of strong operations in a new strong prefix for round k do not create cycles because strong operations are always stored according to a proposal message. From Lemmas 10 and 9, this contains no duplicates. Merging two histories does not create duplicates unless the merged histories have duplicates, and this would imply that some other prior history contains duplicates, a contradiction. □

Lemma 12. *If before a time t a process p_i abdelivers a message m_i and a process p_j abdelivers a message m_j, then some of the two processes abdelivers both m_i and m_j before t.*

Proof. Assume by contradiction that this would not be the case. This implies that abcast never satisfies uniform agreement and total order in runs where $\mathcal{D} \in \Diamond S$ and a majority of correct processes is present. In fact, if uniform agreement holds, p_i and p_j will abdeliver m_i and m_j at some time after t. Therefore p_i will deliver m_i before m_j and p_j will do the opposite. This represents a violation of total order. $\square$

Lemma 13. *For each processes p_i and p_j, if p_i stores $P_i = P'$ and $k_i = k'$ upon abdelivering a message m and p_j abdelivers m then p_j stores $P_j = P'$ and $k_j = k'$ upon receiving m.*

Proof. The proof is by induction on the delivery order of m at p_i. In the base case, all processes p_i have initially the same value of $P_i = \perp$. Let m' be the last message abdelivered by p_i prior to m. For the inductive step, if p_i and p_j abdeliver m' they they both store $P_i = P_j = P_{prev}$ and $k_i = k_j = k_{prev}$ upon abdelivering m'. Assume p_i stores $P_i = P$ and $k_i = k'$ upon abdelivering m and p_j abdelivers m. From Lemma 12, when p_j abdelivers m, it has also already abdelivered every message preceding m in the total order of abcast, so it has abdelivered m'. Upon abdelivering m', p_j stores $P_j = P_{prev}$ and $k_j = k_{prev}$. The next values of P_j and k_j are only determined upon abdelivering m and are only dependent on the value of m and on the previous value of P_j and k_j. Therefore, p_j also stores $P_j = P'$ and $k_j = k'$ upon receiving m. $\square$

Lemma 14. *For each k', processes p_i and p_j and times t_i and t_j, if p_i stores $P_i = (*, *, k', h)$ at time t_i and p_j stores $P_j = (*, *, k', l)$ at time t_j, then $h = l$.*

Proof. By contradiction, assume $h \neq l$ for some times t_i and t_j. p_i must have set $P_i = (*, *, k', h)$ upon abdelivering a PROP$(*, *, k')$ message m_i with $k' = k_i$ before t_i and p_j must have set $P_j = (*, *, k', l)$ upon abdelivering a PROP$(*, *, k')$ message m_j with $k' = k_j$ before t_j. From Lemma 12 some process, assume wlog p_j, has received both m_i and m_j. Also assume wlog that m_i is abdelivered by p_j before m_j in the total order. From Lemma 13, p_j stores $P_j = P_i$ upon receiving m_i and has $k_j = k_i = k'$. After this time and before p_j receives m_j, p_j must have set $P_j = \perp$ because it has changed the third field of P_j. This follows from the definition of *from-round-winner*. Whenever P_j is set to $\perp$, however, k_j is set to $k_j + 1 = k' + 1$. From predicate *from-round-winner*, process p_j will thus never set P_j to a value $(*, *, k', l)$, a contradiction. $\square$

Lemma 15. *If a process p_i delivers its local history and stores $P_i = P' = (*, *, k', i)$ upon abdelivering a message m and a process p_j stores $P_j = (*, *, k', *)$ upon abdelivering m or afterwards, then $P_j = P'$.*

Proof. It follows from *proposal-stable* that is p_i delivers its local history when $P_i = P'$, then p_i does this upon abdelivering a PROP$(*, *, k')$ message

m from itself. d_i is set to k' upon the abdelivery of m. After this time, p_i only abcasts PROP$(*, *, k_i)$ messages with $k_i > d_i = k'$. From Lemma 13, process p_j stores $P_j = P'$ upon abdelivering m. After this time, p_j modifies P_j only if it abdelivers a PROP$(*, *, k')$ from p_i, but no such messages is received after m because of the FIFO property of abcast, or if p_j sets P_j to $\perp$, but then p_j sets k_j to $k' + 1$ and, by definition of *from-round-winner*, will never set P_j to $(*, *, k', *)$ again. □

Lemma 16. *If two processes p_i and p_j store longest strong prefixes π_i and π_j for round k, then $\pi_i = \pi_j$ and every $\varphi_{k'}$ stored by any process for round $k' < k$ is a prefix of π_i and π_j*

Proof. The proof is by induction on k. The property trivially holds for $k = 0$ when the strong prefixes of all processes are empty.

For $k > 0$, if by contradiction p_i and p_j would store different strong prefixes π_i and π_j for round k upon receiving a PUSH message, then some other process would have directly stored those prefixes. Therefore, the problem is reduced to showing the thesis if p_i and p_j *directly* store π_i and π_j. Assume by contradiction that processes p_i and p_j directly store different strong prefixes π_i and π_j upon abdelivering PROP$(*, *, k)$ or CLOSE-RND(k) messages m_i and m_j. From Lemma 10, p_i and p_j have stored exactly one strong prefix, φ^i_{k-1} and φ^j_{k-1} respectively $d_i = d_j = k-1$ upon abdelivering these messages. By induction, $\varphi^i_{k-1} = \varphi^j_{k-1} = \varphi_{k-1}$ is the current longest strong prefix stored by both p_i and p_j immediately before abdelivering m_i and m_j.

Consider now two different cases. The first case is that at least one of m_i and m_j is a PROP(H, S, k). The second is that both m_i and m_j are CLOSE-RND(k) messages.

If at least one of p_i and p_j, say wlog p_i, stores π_i upon abdelivering a PROP(H', S, k) message m_i, then from *prop-stable* this was sent by p_i and, as it was shown, φ_{k-1} is a prefix of H'. π_i is then obtained by p_i by appending elements of S to H' in $<_D$ order. Since φ_{k-1} is a prefix of H', it is also a prefix of π_i. Let $P = (H', S, k, i)$ the value of P_i stored by p_i when π_i is stored.

From *prop-stable*, p_j does not stores π_j before abdelivering m_i. Assume by contradiction that m_j precedes m_i in the total order of abcast. p_j would have stored $k_j = k + 1$ upon abdelivering m_j. Since p_i abdelivers m_i which follows m_j in the total order, it follows from Lemma 12 that p_i abdelivers m_j before m_i. From Lemma 13, p_i would also set $k_i = k + 1$, upon receiving m' and, from *proposal-stable*, it would thus not store a strong prefix for round k upon receiving m_i, a contradiction. Therefore, m_j follows m_i in the total order of the abcast.

From Lemma 13, p_i stores $P_j = (H', S, k, i)$ upon abdelivering m_i. From Lemma 15, $P_j = P$ upon abdelivering m_j. From *proposal-stable*, m_j can not be a PROP$(*, *, k)$ message so it must be a CLOSE-RND(k). When m_j is abdelivered by p_j, p_j builds the same strong prefix $H_{new} = \pi_i$ as stored by p_i since $P_j = P$. π_j is obtained by merging the current local history of p_j with H_n. From Lemma 10, $d_j = k - 1$ so $k > d_j$ and the merge returns $\pi_j = \pi_i$. Also from Lemma 10, φ_{k-1} is a prefix of π_j and of π_i, so the result of the merge is the longest strong prefix stored by p_j. This contradiction concludes the proof for the first case.

Consider now the second case where both p_i and p_j store π_i and π_j upon abdelivering CLOSE-RND(k) messages m_i and m_j. Assume wlog that m_i precedes m_j in the total order of abcast. Let (H', S, k, h) be the value of P_i when before m_i is abdelivered. From *round-winner*, P_i was set to a value $(*, *, k, h)$ for the first time only after p_i abdelivers a PROP$(*, *, k)$ message m' from process p_h. m_i is the first CLOSE-RND(k) message abdelivered after m_j in the total order of abcast. If this would not be the case, p_i would have set $k_i > k$ and would not have stored a strong prefix upon abdelivering m_i, a contradiction. p_i obtains $H_{new} = \pi_i$ by appending operations of S onto H' in $<_D$ order. From Lemma 9 and by the induction hypothesis, φ_{k-1} is a prefix of the local history H' stored by a process p_h. This implies that φ_{k-1} is a prefix of π_i so π_i is a new longest strong prefix of p_i. From Lemma 12 and total order of abcast, p_j also delivers m' before m_i and m_i before m_j. From Lemma 13, p_j also sets P_j to $(*, *, k, h)$ for the first time upon abdelivering m'. It has been already shown that m_i is the first CLOSE-RND(k) message which is abdelivered after m'. From Lemma 13, p_j also stores a strong prefix π_j for round k and builds $\pi_j = \pi_i$ upon abdelivering m_i. This is the new longest prefix since φ_{k-1} is a prefix of π_i. This is the final contradiction. □

Lemma 17. *For each processes p_i and p_j and times t_i and t_j, if π_i is a strong prefix of $H(i, t_i)$ and π_j is a strong prefix of $H(j, t_j)$ then π_i and π_j are compatible.*

Proof. When a process p_i stores a strong prefix for round k, it sets $d_i = k$ and stores no other strong prefixes for rounds $k' \leq d_i$ afterwards. Therefore, the result follows directly from Lemma 16. □

Lemma 18. *For each process p_i and times t and t', if $t' > t$ and π is a strong prefix of $H(i, t_i)$ then π_j is a strong prefix of $H(i, t')$.*

Proof. The result directly follows from Lemma 16 if $p_i = p_j$. □

Lemma 19. *For each times t and t_i and correct processes p_i and p_j and for each operation op submitted by any process before t and included in $H(i, t_i)$, if $t' \geq ord(t)$ then $op \in H(j, t')$*

Proof. Assume by contradiction that there exists an operation *op* submitted before t and included in $H(i, t_i$ such that *op* not in H_j. Since *op* not in $H(j, t')$ and $t' \geq ord(t)$, p_j never includes *op* into its history by definition of $ord(t)$.

Assume that *op* is a weak operation. Since *op* is stored by p_i, p_i eventually sends a PUSH$(H, *)$ message including $op \in H$ to p_j. Since p_i and p_j are correct, p_j eventually receives the PUSH message and calculates its new local history as a merge between H and its previous local history. The resulting history contains *op*, a contradiction.

Assume now that *op* is strong and let k' be the round number where p_i stores the first strong prefix π_i including *op*. After storing π_i, p_i stores $d_i \geq k'$. If p_j stores a strong prefix for round $k'' \geq k'$, it also stores π_i from Lemma 16, a contradiction. Therefore, p_j never stores a strong prefix for a round $k'' \geq k'$ and thus never sets $d_j \geq k'$. However, p_i eventually sends a PUSH$(*, d_i)$ message with $d_i \geq k'$ to p_j. Since both p_i and p_j are correct, p_j eventually receives the PUSH message. After the subsequent merge, p_j stores $d_j \geq k'$, a contradiction. □

Lemma 20. *If there exists a time t_{ld} when p_{ld} is perpetually trusted, then for each $t' \geq ord(ord(t_{ld}))$ and for each correct process p_i, $H(i, t')$ is a subset of $H(ld, t')$.*

Proof. The proof is that $H_i = H(i, t')$ is a subset of $H_{ld} = H(ld, t')$ and is by contradiction. Assume that there exists an operation *op* submitted by a process p_j such that $op \in H_i$ and $op \notin H_{ld}$. If *op* is submitted before t_{ld}, thesis follows from $t' \geq ord(t_{ld})$ and Lemma 19. Therefore, *op* is submitted after t_{ld}.

Consider two cases. If *op* is a weak operation, p_j trusts p_{ld} when *op* is submitted and sends a WREQ msg only to p_{ld}. p_{ld} is the first process to add *op* to its history and all other processes store *op* in their history after directly or indirectly merging their history with the one of p_{ld}. Therefore, if $op \in H_i$ then $op \in H_{ld}$.

If *op* is a strong operation, let k' be the round number where p_i stores the first strong prefix π_k including *op*. Since *op* is submitted after t_{ld} and p_{ld} is perpetually trusted, it follows from *must-propose-prefix* that p_{ld} is the only process which abcasts a PROP$(*, *, k')$ message. This implies that no process $p_j \neq p_{ld}$ ever sets $P_j = (*, *, k', j)$. Therefore, any process $p_j \neq p_{ld}$ that directly stores π_k for round k' does it upon abdelivering a CLOSE-RND(k') message m. Since p_{ld} is the perpetual leader, no process $p_j \neq p_{ld}$ abcasts a CLOSE-RND(k') message. Therefore m is sent by p_{ld} after having stored π_{ld} in its history. From Lemma 16, π_{ld} is equal to π_k and thus includes *op*. Any other process, like p_i, which stores π_i for round k' does it after p_{ld}. This implies that if $op \in H_i$ then $op \in H$. □

Lemma 21. *For each time t and $t' \geq t$, if $op <_{i,t} op'$, op and op' are not in a strong prefix of $H(i,t)$ or of $H(i,t')$ and $op <_D op'$, then $op <_{i,t'} op'$*

Proof. Since operations are never removed from a history and $op <_{i,t'} op'$, p_i stores op and op' for any time $t' \geq t$. Assume by contradiction that for some time $t'' \geq t$, p_i orders op' before op for the first time in its local history. The order of two operations is changed in a local history only by making a merge. However, any merged history always keeps $op <_{i,t''} op'$ as $op' <_D op$ and op and op' are not in a strong prefix of $H(i,t'')$. □

Lemma 22. *For each time t, if p_i and p_j are correct processes, $op <_{i,t_i} op'$ and $op' <_{j,t_j} op$ op and op' are submitted before t, op and op' are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$ and $op' <_D op$ in the deterministic order, then $t_i < ord(t)$.*

Proof. Assume by contradiction $t_i \geq ord(t)$. Assume that p_i receives at time $t' \leq ord(t)$ a PUSH$(H_p, *)$ message m sent by p_j at time $t'' \leq t'$ with a history containing $op' <_{H_p} op$. Neither op nor op' are in the strong prefix of $H(i,t'')$ or H_p because otherwise they would also be in a strong prefix of the local history of p_i at time $ord(t) \geq t'$ from the definition of the merge operation and from LEMMA 18. When m is received, p_i merges the H_p in its local history. The resulting history orders $op' < op$ as $op' <_D op$ and as op and op' are not in a strong prefix of $H(i,t'')$ or of H_p. From Lemma 21, $op' <_{i,t_i} op$ for each time $t_i \geq t''$ so also for each time $t_i \geq ord(t)$, a contradiction.

It remains now to be shown that p_i receives a PUSH$(H_p, *)$ message m from p_j with a history containing $op' < op$ at a time $t'' \leq ord(t)$. Assume p_i does not receive any history where op' precedes op before $ord(t)$. By definition of $ord(t)$ and since op and op' are both submitted before t, p_i never receives a history containing where op' precedes op. Since $op' <_{j,t_j} op$, process p_j eventually send a PUSH$(H_p, *)$ message to p_i. From Lemma 21, $H(j,t')$ still orders op' before op and so does H_p. Since p_i and p_j correct, p_i eventually receives H_p, a contradiction. □

Lemma 23. *For each time t, if $t_i, t_j \geq ord(t)$, p_i and p_j are correct processes, op and op' are submitted before t and are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$, then it never holds $op <_{i,t_i} op'$ and $op' <_{j,t_j} op$.*

Proof. Assume by contradiction that $op <_{i,t_i} op'$ and $op' <_{j,t_j} op$. If $op <_D op'$ it follows from Lemma 22 that $t_j < ord(t)$, a contradiction. Similarly, if $op' <_D op$ then $t_i < ord(t)$, a contradiction. □

Lemma 24. *For each time t, if $t_i, t_j \geq ord(ord(t))$, p_i and p_j are correct processes, op is submitted before t, $op' <_{i,t_i} op$ and op and op' are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$, then $op' <_{j,t_j} op$.*

Proof. Assume by contradiction that $op' \not<_{j,t_j} op$. Also, assume that op' is submitted before $ord(t)$. Since p_i stores op and op', p_j stores op and op' at time $t_j \geq ord(ord(t))$ from Lemma 19. This implies that $op <_{j,t_j} op'$. Since both op and op' are submitted before $ord(t)$ and are not in the strong prefix of $H(i,t_i)$ or $H(j,t_j)$, a contradiction follows from Lemma 23.

It is now necessary to show that op' is submitted before $ord(t)$. op and op' are weak operations because are not included in a strong prefix. There are two ways for p_i to store op' before op. p_i can directly append op after op' in its history or can merge its local history with another history H such that $op' <_H op$ and contained in a PUSH$(H,*)$ message. In both cases, some process p_k has directly appended op after op'. By definition of $ord(t)$ and since p_k stores op, op these operations were already stored by p_k at time $ord(t)$. Since op is appended by p_k in its local history after op', op' was already stored by p_k at time $ord(t)$. Therefore, op' is submitted before $ord(t)$. □

Lemma 25. *For each pair of operations op and op', times t_i and t_j and correct processes p_i and p_j if there exists a time t_{ld} when p_{ld} is perpetually trusted, $t_i, t_j >= ord(ord(t_{ld}))$, op and op' are not in a strong prefix of $H(i,t_i)$ or $H(j,t_j)$ and $op' <_{i,t_i} op$ and $op \in H(j,t_j)$ then $op' <_{j,t_j} op$.*

Proof. If op is submitted before t_{ld}, the result directly follows from Lemma 24. Therefore, op and op' are submitted after t_{ld}.

If op or op' are in a strong prefix, since p_{ld} is the only process which trusts itself after t_{ld} and from *must-propose-new-prefix*, it follow stat p_{ld} is the only process which abcasts PROP$(H,S,*)$ messages with op or op' in $H \cup S$. Else, p_{ld} is the first process to establish an order for op and op'. In both cases, if a process p_i stores op' before op, this is the order established by p_{ld}. Therefore, each process p_j storing op also lets it precede by op in its local history.

The last remaining case is the one where op' is submitted before t_{ld} and op is submitted after t_{ld}. From Lemma 19 and the fact that p_i stores op', p_j stores op' before $ord(t_{ld})$. Also, p_j stores op by hypothesis, so p_j has ordered op and op' at time t_j. Assume by contradiction that $op <_{j,t_j} op'$. This and $op' <_{i,t_i} op$ would contradict Lemma 24. □

Lemma 26. *For any pair of operations op and op', times t' and t'', and correct process p_i if there exists a time t_{ld} when p_{ld} is perpetually trusted $t', t'' \geq ord(ord(ord(t_{ld})))$ and $op <_{ld,t'} op'$, then $op' \not<_{i,t''} op$.*

Proof. Assume by contradiction that $op' <_{i,t''} op$. If op (resp. op') is strong, a contradiction directly follows from Lemma 17 and $op' <_{i,t''} op$ (resp. $op <_{ld,t'} op'$). Therefore, op and op' are weak.

If op and op' are not in a strong prefix, a contradiction follows directly from Lemma 25. Therefore, both operations are in a strong prefix.

Let k be the minimum round number such that op or op' are in a strong prefix π of $H(ld, t')$ or $H(i, t'')$. π either includes op but not op', or op' but not op, else Lemma 17 would be violated by p_{ld} or p_i. Assume that π includes op' but not op. The argument in case π includes op but not op' is similar. The main differences are discussed below.

Assume that π has been submitted before $ord(ord(t))$. Since p_i or p_{ld} have stored π, all other correct processes do the same before $ord(ord(ord(t)))$ from Lemma 19. Therefore, p_{ld} stores op' before op at time $t' > ord(ord(ord(t_{ld})))$ but this is inconsistent with π, a contradiction of Lemma 18. In case π only includes op', a similar contradiction is built with p_i.

It is now necessary to show that π has been submitted before $ord(ord(t))$. By definition, π is built by a process after abdelivering a PROP(H', S', k) message m from a process p_h, and is the result of appending the strong operations of S' onto H'. Since op' is weak, $op' \in H'$. Let t_h be the time when p_h sends m. Since op' is in H' then $op' \in H(h, t_h)$. By definition of k, neither op nor op' are in a strong prefix of $H(h, t_h)$. Assume by contradiction that $t_h \geq ord(ord(t_{ld}))$. It follows from this, Lemma 25, $op' \in H(h, t_h)$ and $op <_{ld,t''} op'$ that $op <_{h,t_h} op'$. Therefore H', and thus the strong prefix π too, would include op and op', a contradiction. In case π includes op but not op', a contradiction would follow from Lemma 25 and $op' <_{i,t_i} op$ since π would contain op and op'. This implies that $t_h < ord(ord(t_{ld}))$ so π has been submitted before $ord(ord(t_{ld}))$. □

Lemma 27. *If there exists a time t_{ld} when a process p_{ld} is trusted by all processes, then there exists a time t such that for each $t' \geq t$ and for each correct process p_i it holds that $H(i, t)$ is a prefix of $H(i, t')$.*

Proof. Let $H(i, t)$ be the history stored by process p_i at time t. By contradiction, assume that $t = ord(ord(ord(t_{ld})))$, and let $t_m \geq t$ be the minimum time such that $H = H(i, t)$ is not a prefix of $H_m = H(i, t_m)$.

H is a subset of H_m and H_m is a subset of $H(ld, t_m)$. The first fact follows from the fact that histories are modified by appending operations or by merging and that merges return the union of the merged histories. The second follows from Lemma 20. From Lemma 26, both H and H_m order their operations as in $H(ld, t_m)$, so H is a prefix of H_m, a contradiction. □

Lemma 28. *If a correct process p_{ld} which is eventually permanently trusted by $\Omega_{\mathcal{D}}$ abcasts a PROP$(*, *, k)$ message and eventually stops modifying H_{ld} until $k_{ld} > k$, and if $\Omega_{\mathcal{D}} \in \Omega$ and a majority of correct processes exists, then eventually p_{ld} sets $k_{ld} > k$ and $Q_{ld} = \perp$.*

Proof. The proof is by contradiction. By hypothesis, p_{ld} abcasts a PROP$(*, *, k)$ message. Since $\Omega_{\mathcal{D}} \in \Omega$ and a majority of correct processes exists, abcast terminates. This and the fact that p_{ld} is correct implies that some process will be the winner of round k by having its proposal abdelivered.

If p_{ld} is the winner of round k, it sets $P_{ld} = (*, *, k, ld)$ and $Q \neq \perp$. If p_{ld} later abdelivers a CLOSE-RND(k) message, it sets $k_{ld} > k$ and $Q_{ld} = \perp$, a contradiction. Therefore, p_{ld} never abdelviers a CLOSE-RND(k) message so, from validity of abcast, p_{ld} never abcasts such a message. This implies that $\Omega_{\mathcal{D}}$ at p_{ld} always outputs ld. From *must-propose-new-prefix*, p_{ld} keeps sending proposal messages whenever its local history is modified. From validity of abcast, process p_{ld} abdelivers all the proposal messages that it abcasts. By hypothesis, p_{ld} eventually stops adding operations to its local history H_{ld} during round k. Therefore, process p_{ld} will eventually abdeliver a PROP$(H', *, k)$ message sent from itself with $H' = H_{ld}$. It will therefore abcast a CLOSE-RND(k) message, a contradiction.

If $p_j \neq p_{ld}$ is the winner of round k, p_{ld} sets $P_{ld} = (*, *, k, j)$. It is sufficient to show that p_{ld} abcasts or abdelivers a CLOSE-RND(k) message to reach a contradiction like in the previous case. Therefore p_{ld} never abdelivers a CLOSE-RND(k) message from the winner p_j. This implies that eventually *suspect-round-winner*$_{ld}$ will hold since p_{ld} is the only process which is permanently trusted by $\Omega_{\mathcal{D}}$. Therefore, p_{ld} will abcast a CLOSE-RND(k) message, a contradiction. □

Lemma 29. *If a process p_i stores a new history H_n by merging its local history H and another history H' and both H and H' satisfy properties (C1) and (C2) of causal consistency, then H_n satisfies (C1) and (C2)*

Proof. It is trivial that H_n satisfies (C1) since H_n is the union of H and H'. For (C2), let M be the result of the merge and assume by contradiction that $o <_C o'$ but $o' <_M o$. Since M stores o', one of H and H', say H, stores o'. From (C1), H stores o too. From (C2), $o <_H o'$. Assume that o and o' are not in a strong prefix π of H or H'. Both o and o' are therefore weak operations. From the merge procedure it follows that if $o' <_M o$ and $o <_H o'$ then $o' <'_H o$. H' thus violates (C2), a contradiction.

Next, it is shown that o and o' are not in a strong prefix π of H or H'. Assume by contradiction that they are. From Lemmas 17 and 18 and the fact that M is stored by a process as new strong prefix, π is a prefix of M.

If $o \in \pi$ then either $o' \notin \pi$ or $o <_\pi o'$ since (C1) and (C2) are not violated in π. For the same reason, if o' is in π then $o <_\pi o'$. In all these cases, since π is a prefix of M then $o' \not<_M o$, a contradiction. $\square$

Theorem 5. *Causal consistency is satisfied.*

Proof. Assume that a process p_i is the first process to violate (C1) or (C2) at time t. A process violated these properties only when it modifies its local history. If p_i appends an operation it has submitted to its local history, a contradiction directly follows from the fact that the prior local history satisfies (C1) and (C2).

If p_i violates (C1) or (C2) upon receiving a PUSH or ORD message m at time t, the new history of p_i is the merge between the old history of p_i and the history contained in the message. Both merged histories are local histories of processes at a time preceding t so they satisfy (C1) and (C2). A contradiction follows from Lemma 29.

Consider now the case when p_i violates (C1) or (C2) upon receiving a WREQ(H, o) or SREQ(H, o) message m at time t. p_i merges its history with H and, similar to the previous case, the result satisfies (C1) and (C2). Also, H contains all operations o' such that $o' <_C o$. Appending o to the new local history of p_i preserves (C1) and (C2).

The last case is that p_i violates (C1) or (C2) upon abdelivering a PROP or CLOSE-RND message. If the local history of p_i is modified upon receiving these messages, then p_i stores a new strong prefix for round k and sets $P_j = (H, S, k, h)$. Let H_n be the result of appending all operations of S onto H in a deterministic order. Since the previous local history of p_i satisfies (C1) and (C2), it is sufficient from Lemma 29 to show that H_n satisfies these properties.

If p_i has set $P_i = (H, S, k, h)$ then a process p_h has abcast a PROP(H, S, k) message. For each strong operation $o \in S$, p_h has received from the proposer processes histories including all operations o' such that $o' <_C o$. H is the local history of p_h has merged all these histories and, from Lemma 29, satisfies (C1) and (C2) and includes all operations causally dependent on operations in S. From Lemmas 11 and 16, all the operations in S has not yet been stored by any other process for any other round. This implies that none of the operations of S is causally dependent on each other, so H_n satisfies (C1) and (C2). $\square$

Theorem 6. *Nontriviality, set stability, strong prefix stability, prefix consistency, strong prefix consistency are always satisfied.*

Proof. This proof shows that all properties of Eventual Consistency are met. For each process p_i and time t, the properties of $S(i, t)$ are shown for local histories $H(i, t)$. Since only the content of local histories is ever delivered, and since local histories are delivered whenever they are modified, this is equivalent to show the properties for delivered sequences.

Nontriviality: Is trivial from the algorithm and from Lemma 11.

Set stability: Directly follows from the fact that histories are modified either by appending operations or from merges. The latter operation returns the union of the merged histories, so no operation is removed from a history.

Strong prefix stability: Directly follows from Lemma 18.

Strong prefix consistency: Directly follows from Lemma 17.

Prefix consistency: P_t is defined as follows. For each operation op stored by a correct process, let $t(op)$ be the time when op is submitted and $p(op)$ the first correct process storing op. P_t includes all operations stored by a correct process such that $t \geq ord(ord(t(op)))$, as well as the prefix including op in $H(p(op), t(op))$, in the order of $H(p(op), t(op))$.

The first step is showing that P_t satisfies (C1) and is a sequence. From Lemma 19, all operations that are submitted before $t(op)$ and that are stored by a correct process are stored by each correct process at time $t' \geq ord(t(op))$. From strong prefix consistency and strong prefix stability, the longest strong prefix of P_t is a prefix of $H(i, t')$ for each i and $t' \geq t$. From Lemma 25, the prefix preceding each remaining operation of P_t in $H(i, t')$ is equal at each correct process p_i at time $t' \geq t$ since $t = (ord(ord(t(op)))$, so P_t is a prefix of each $H(i, t')$ with $t' \geq t$.

(C2) can be shown easily because, from (C1), P_t and $P_{t'}$ are both prefixes of H(i, t') for each i. Also, each operation of P_t is included in $P_{t'}$ by definition since $t \leq t'$. Therefore, P_t is a prefix of $P_{t'}$.

As for (C3), it follows from Liveness that all operations invoked by a correct process are eventually stored by all other processes. From Lemma 19, all operations stored by a correct process are eventually stored by each correct process, so all operations stored by a correct process are included in some P_t for some t. □

Theorem 7. *Eventual Stability is satisfied if $\mathcal{D} \in \Diamond S$.*

Proof. Eventual stability after for some t follows from Lemma 27. □

Theorem 8. *Each weak operation w submitted by a correct process is eventually stored by each correct process in its local history.*

Proof. Assume a correct process p_i submits a weak operation w and some correct process p_j never adds it to its history. Let ld be value of $\Omega_{\mathcal{D}}$ when the submit event occurs. The operation w is reliably sent to p_{ld} in a WREQ message m.

If p_{ld} suspected by $\Omega_{\mathcal{D}}$, p_i appends w to its local history. Eventually p_i sends a PUSH(H, d) message with w in H. Since p_i and p_j are both correct, the PUSH message is eventually delivered. p_j then either adds w into its history or w is already in its history. Therefore, since by contradiction p_i never delivers w, $\Omega_{\mathcal{D}}$ never suspects p_{ld}. By the strong completeness of $\mathcal{D}$, this implies that p_{ld} is correct. The WREQ message m is thus eventually delivered by p_{ld}.

If *wait-consensus*$_{ld}$ is false when m is received by p_{ld}, or it is true, and thus w is included in W_{ld}, but it eventually becomes false, and thus *stop-waiting-consensus*$_{ld}$ holds, p_{ld} merges the history contained in m with its own, and the resulting H_{ld} contains w. After this merge, p_{ld} eventually sends a PUSH message containing w to all correct processes, which eventually receive it and store w in their local history, a contradiction. Therefore, *wait-consensus* is always true. Therefore, it always holds that $Q_{ld} \neq \perp$ and that T_{ld} is a majority quorum equal to the current set $TS_{ld} \setminus \mathcal{D}$.

If a majority of correct processes does not exist, then eventually $|\mathcal{D}_{ld}| \geq \lceil n/2 \rceil$ for strong completeness so $|TS_{ld} \setminus \mathcal{D}| < \lceil n/2 \rceil$, a contradiction. Therefore, there exists a majority of correct processes. From $Q_{ld} \neq \perp$, p_{ld} has sent a PROP$(*, *, k)$ message for some $k = k_{ld}$. If $\Omega_{\mathcal{D}} \in \Omega$, it follows from Lemma 28 that eventually $Q_{ld} = \perp$ and thus *wait-consensus*$_{ld}$ stops holding, a contradiction. Therefore, $\Omega_{\mathcal{D}} \notin \Omega$.

Since *wait-consensus* always holds, it always holds that $|T_{ld}| > n/2$ and $T_{ld} = TS_{ld} \setminus \mathcal{D}$. From the strong completeness of $\mathcal{D}$, $TS_{ld} \setminus \mathcal{D}$ eventually only includes the ids of correct processes. Since $T_{ld} = TS_{ld} \setminus \mathcal{D}$ holds forever, T_{ld} contains the indexes of a majority of correct processes which permanently trust p_{ld}. Therefore, $\Omega_{\mathcal{D}}$ satifies the quorum property so $\Omega_{\mathcal{D}} \in \Omega_Q$. Since there exists a majority of correct processes, Ω and Ω_Q are equivalent from Lemma 8. This implies that $\Omega_{\mathcal{D}} \in \Omega$, a contradiction. □

Theorem 9. *If a correct process p_i submits a strong operation s, there exists a majority of correct processes, and either $\mathcal{D} \in \Diamond P$ or $\mathcal{D} \in \Diamond S$ and eventually no new weak operation is submitted, then each correct process eventually stores s in their history.*

Proof. Assume by contradiction that a correct process p_i submits a strong operation s and there exists a correct process p_j which never stores s in its history.

Let p_{ld} be the correct leader which is eventually perpetually trusted by $\Omega_{\mathcal{D}}$. If p_i or p_{ld} ever store s in their history, then a contradiction is now shown. Let k' be the round when p_i or p_{ld} first store a strong prefix π including s. Each other correct p_j will eventually receive a PUSH(H, d) message from p_i or p_{ld} with s in H and $d \geq k'$. By Lemma 16, if p_j never stores π then it never stores a strong prefix for round k' so $d_j < k'$. When the PUSH message is received then the result of the merge has π as strong prefix, a contradiction.

Neither p_i nor p_{ld} thus ever store s in their local history. After s is submitted, p_i sends SREQ$(*, *, s)$ to all processes. Since p_i and p_{ld} are correct, this message is eventually received by p_{ld}. When this happens, p_{ld} adds s to N_{ld}. However s is never added in H_{ld} of p_{ld} by contradiction. This implies that s is always in $N_{ld} \setminus H_{ld}$.

Let k_s be the current value of k_{ld} when s is received by p_{ld}. For each value $k \geq k_s$ of k_{ld}, eventually p_{ld} either sets $k_{ld} = k + 1$, and thus $Q_{ld} = \perp$ too, or it abcasts a PROP$(*, S, k)$ message with $s \in S$. This follows by simple induction on the value of k_{ld} since $N_{ld} \setminus H_{ld}$ always includes s, since p_{ld} eventually trusts itself permanently, and from *must-propose-new-prefix*. Assume that eventually p_{ld} sets $k_{ld} = k + 1$ in the both the aforementioned cases. Since p_{ld} is the permanent leader, it follows from *must-propose-new-prefix* that there exists a round k' such that p_{ld} is the only process abcasting a proposal messages PROP$(*, S, k')$ for k'. Furthermore, abcast terminates since $\mathcal{D} \in \Diamond S$ implies that $\Omega_{\mathcal{D}} \in \Omega$ and since a majority of correct processes exists. Since p_{ld} is correct, if follows from validity of abcast that it abdelivers it proposal message and, since this is the only proposal for k', that p_{ld} is the winner of round k'. Since eventually $k_{ld} = k' + 1$, this implies that p_{ld} eventually stores its proposed strong prefix for round k'. This strong prefix includes s, a contradiction.

It is now shown that if p_{ld} abcasts a PROP$(*, *, k)$ message then eventually $k_{ld} = k + 1$. This would follow from Lemma 28 if p_{ld} would eventually stop modifying its local history H_{ld} until $k_{ld} = k + 1$. Assume by contradiction that p_{ld} modifies H_{ld} infinitely often and that k_{ld} is always equal to k. A contradiction is easy to see if eventually no weak operation is submitted. Therefore, it must hold that $\mathcal{D} \in \Diamond P$ and that a majority of correct processes exists. In this case, infinitely many weak operations are received by p_{ld} and inserted in H_{ld}. From *must-propose-new-prefix*, this implies that the leader abcasts infinitely many PROP$(*, *, k)$ messages since $Q \neq \perp$ after sending the first PROP$(*, *, k)$ message. However, since p_{ld} is perpetually trusted, eventually every correct process sends a TRUST(p_{ld}) message to p_{ld} as last trust message. This implies that the trust set TS_{ld} of p_{ld} eventually does not change any longer. Also, it follows from $\mathcal{D} \in \Diamond P$ that eventually $\mathcal{D}$ outputs exactly the ids of the faulty processes, so eventually $TS_{ld} \setminus \mathcal{D}_{ld} > n/2$ holds forever and $\mathcal{D}$ stops changing. Whenever a new proposal message is abcast by p_{ld}, T_{ld} is set to be equal to $TS_{ld} \setminus \mathcal{D}$, so eventually T_{ld} is equal to $TS_{ld} \setminus \mathcal{D}$ forever. Therefore, eventually *wait-consensus*$_{ld}$ holds forever and p_{ld} stops modifying H_{ld}, a contradiction. □

Bibliography

[AAL+08] A.S. Aiyer, E. Anderson, X. Li, M.A. Shah, and J.J. Wylie. Consistability: Describing usually consistent systems. In *Proc. of Fourth Workshop on Hot Topics in System Dependability*, 2008.

[ACKL07] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Customizable fault tolerance for wide-area byzantine replication. In *Proc. of the twenty-sixth IEEE International Symposium on Reliable Distributed Systems*, 2007.

[ACKL08] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *In Proceedings of the 38th IEEE/IFIP International Conference on Depe ndable Systems and Networks (DSN '08*, pages 105–114, 2008.

[AEMGG+05] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, 2005.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors (extended abstract). In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 679–690, New York, NY, USA, 1992. ACM.

[AT10] Marcos K. Aguilera and Sam Toueg. Adaptive progress: A gracefully-degrading liveness property. *Distributed Computing*, 22(5–6):303–334, August 2010.

[AW04] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* J. Wiley and Sons, 2004.

[BCvR09] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[Bor05] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[BT83] Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26, New York, NY, USA, 1983. ACM.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[Cas01] Miguel Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, MIT, January 2001.

[CC10] Christian Cachin and Christian" Cachin. Yet another visit to paxos. Technical report, IBM Research Zürich, 2010.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[Chu98] F. Chu. Reducing ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 67:289–293, 1998.

[CKL$^+$09] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, New York, NY, USA, 2009. ACM.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[CML$^+$06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. *SIGOPS Oper. Syst. Rev.*, 41(6):189–204, 2007.

[CNV04] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.

[Con02] Cristian Constantinescu. Impact of deep submicron technology on dependability of vlsi circuits. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 205–209, Washington, DC, USA, 2002. IEEE Computer Society.

[CRS$^+$08] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *Very Large Data Bases Conference*, 2008.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CVN02] Miguel Correia, Paulo Verissimo, and Nuno Ferreira Neves. The design of a cotsreal-time distributed security kernel. In *EDCC-4: Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 234–252, London, UK, 2002. Springer-Verlag.

[CWA$^+$09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI'09: Proceedings of the 6th*

USENIX symposium on Networked systems design and implementation, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[Das] Amazon Web Services Service Health Dashboard. Amazon s3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.

[DCGN03] Michael Dahlin, Bharat Baddepudi V. Chandra, Lei Gao, and Amol Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking (TON)*, 11(2):300–313, April 2003.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[DGFG$^+$04] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346, New York, NY, USA, 2004. ACM.

[DGV04] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous byzantine consensus. Technical Report LPD-REPORT-2008-08, EPFL, 2004.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, 2007.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[FGK06] Felix C. Freiling, Rachid Guerraoui, and Petr Kouznetsov. The failure detector abstraction. Technical report, Universität Mannheim / Institut für Informatik, 2006.

[FGL$^+$96] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 300–309, New York, NY, USA, 1996. ACM.

[FL81] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14:183–186, 1981.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM Symposium on Operating Systems Principles*, 2003.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[GKQV10] Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 bft protocols. In *Proceedings of Eurosys 2010*, 2010.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[Gro] Trusted Computing Group. Trusted computing group specifications. www.trustedcomputinggroup.org.

[Gue00] Rachid Guerraoui. Indulgent algorithms (preliminary version). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 289–297, New York, NY, USA, 2000. ACM.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Ben Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, 2010.

[HW87] Maurice P. Herlihy and Jeanette M. Wing. Specifying graceful degradation in distributed systems. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 167–177, New York, NY, USA, 1987. ACM.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[KAD$^+$07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles*, pages 45–58, 2007.

[KD96] Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Proceedings of fifteenth ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, 1996.

[KMMS98] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. *Hawaii International Conference on System Sciences*, 3:317, 1998.

[KPA$^+$03] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, 2003.

[LAC07] Harry C. Li, Lorenzo Alvisi, and Allen Clement. The game of paxos. In *Proceedings of the he 26th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2007.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[Lam03] Leslie Lamport. Lower bounds for asynchronous consensus. In *Proceedings of the Future Directions in Distributed Computing Workshop*, pages 22–23, October 2003.

[Lam05] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[LDLM09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: small trusted hardware for large distributed systems. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, 1992.

[LPS01] Bev Littlewood, Peter Popov, and Lorenzo Strigini. Modeling software design diversity: a review. *ACM Comput. Surv.*, 33(2):177–208, 2001.

[LSP82a] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LSP82b] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[MA06] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[MBTPV06] Carlo Marchetti, Roberto Baldoni, Sara Tucci-Piergiovanni, and Antonino Virgillito. Fully distributed three-tier active software replication. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):633–645, 2006.

[MP91] F. J. Meyer and D. K. Pradhan. Consensus with dual failure modes. *IEEE Trans. Parallel Distrib. Syst.*, 2(2):214–222, 1991.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[PWB07] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andr Barroso. Failure trends in a large disk failure trends in a large disk drive population. In *Proceeding of fifth USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[RCL01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2001. ACM.

[RMS10] O. Rütti, Z. Milosevic, and A. Schiper. Generic construction of consensus algorithms for benign and byzantine faults. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[RS95] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. Technical report, IRISA TR. 968, 1995.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SDM$^+$08] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. Bft protocols under fire. In *Proc. of Fifth USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[SFK$^+$09] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[SG07] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of disk failures in the real world: What does an mttf of disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceeding of fifth USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[SJ10] Marco Serafini and Flavio P. Junqueira. Weak consistency as a last resort. In *Proceedings of the the 4th ACM SIGOPS/SIGACT Workshop on Large Scale Distributed Systems and Middleware*, 2010.

[SJR09] Yee Jiun Song, Flavio P. Junqueira, and Ben Reed. Bft for the skeptics. In *BFTW3 Workshop*, 2009.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[SS07] Marco Serafini and Neeraj Suri. The fail-heterogeneous architectural model. In *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 103–113, Washington, DC, USA, 2007. IEEE Computer Society.

[TP88] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 93–100, 1988.

[TRAR99] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed consistency for shared distributed objects. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 163–172, New York, NY, USA, 1999. ACM.

[TTP$^+$95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.

[Ver06] Paulo Verissimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

[Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[WLG$^+$78] J H WENSLEY, L LAMPORT, J GOLDBERG, M W GREEN, K N LEVITT, P M MELLIAR-SMITH, R E SHOSTAK, and C B WEINSTOCK. Sift - design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66:1240–1255, October 1978.

[WS03] Chris J. Walter and Neeraj Suri. The customizable fault/error model for dependable distributed systems. *Journal of Theoretical Computer Science*, 290:1223–1251, 2003.

[YMV$^+$03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267, New York, NY, USA, 2003. ACM.

[ZBWM08] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[ZPR$^+$07] Lidong Zhou, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Roy Levin, and Chandramohan A. Thekkath. Graceful degradation via versions: specifications and implementations. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 264–273, New York, NY, USA, 2007. ACM.

Curriculum Vitae

Personal Data

Name: Marco Serafini

Date of birth: January 24th, 1979

Place of birth: Arezzo, Italy

Education

1992-1997 *Maturità degree* – 60/60 – Liceo Scientifico Francesco Redi, Arezzo, Italy

1997-2004 *Laurea degree in Computer Science* – 110/110 “Summa cum laude” – Faculty of Mathematics, Physics and Natural Sciences, University of Florence, Italy

2004-2010 *Ph.D. in Computer Science* – Technische Universität Darmstadt, Darmstadt, Germany

Awards

- Recipient of DSN 2007 student scholarship
- Recipient of Graduiertenkolleg (GK) scholarship “Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments“
- Recipient of SOSP and LADIS 2009 student scholarships
- Recipient of Eurosys SOSP 2009 student scholarship
- Recipient of DSN 2010 student scholarship

Patents

- “Method and apparatus for monitoring the status of nodes of a communication network“ - Inventor

Invited Talks

- AUDI AG, Ingolstadt, Germany. “Handling Transient and Intermittent Faults in Embedded Distributed Systems”. September 11, 2006.
- Department of Computer Science, University of Florence, Italy. “From Paxos to Speculative Byzantine Fault Tolerance”. July 24, 2008.
- Hitachi Research Lab, Omika, Japan. “Design, Validation and Verification of a Diagnostic Protocol for Safety Critical System”. December 12, 2008.
- Max Planck Institute for Software Systems (MPI-SWS), Saarbruecken, Germany. “Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas”. December 22, 2009.
- Newcastle University, Newcastle Upon Tyne, UK. “Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas”. March 25, 2010.

Publications

Journal articles

1. Marco Serafini, Peter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstaetter, Fulvio Tagliabo’, Jens Koch, “Application-Level Diagnostic and Membership Protocols for Generic Time-Triggered Systems”, IEEE Transactions on Dependable and Secure Computing (IEEE TDSC) – accepted, to appear
2. Marco Serafini, Andrea Bondavalli and Neeraj Suri, “On-Line Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters”, IEEE Transactions on Dependable and Secure Computing (IEEE TDSC), 4(4), Oct. 2007

Conference articles

1. Marco Serafini and Flavio Junqueira, *Weak Consistency as Last Resort*, in Proc. of ACM SIGOPS/SIGACT Workshop on Large Scale Distributed Systems and Middleware (LADIS), Zürich (CH), 2010.

2. Marco Serafini, Dan Dobre, Matthias Majuntke, Peter Bokor and Neeraj Suri, "Eventually Linearizable Shared Objects", Proc. of ACM Symp. on Principles of Distributed Computing (PODC), 2010.

3. Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke and Neeraj Suri, "Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas", Proc. of IEEE Int'l. Conf. on Dependable Systems and Networks (DSN-DCCS), 2010.

4. Marco Serafini and Neeraj Suri, "Reducing the Costs of Large-Scale BFT Replication", Proc. of Large-Scale Distributed Systems and Middleware (LADIS), 2008.

5. Marco Serafini and Neeraj Suri, "Trust Characterization for Dependable Distributed Systems", Eurosys student seminar, 2008

6. Marco Serafini and Neeraj Suri, "The Fail-Heterogeneous Architectural Model", Proc. of the IEEE Int'l Symp. on Reliable Distributed Systems (SRDS), 2007

7. Marco Serafini, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstätter, Fulvio Tagliabo' and Jens Koch, "A Tunable Add-On Diagnostic Protocol for Time-Triggered Systems", Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks (DSN-DCCS), 2007.

8. Peter Bokor, Marco Serafini and Neeraj Suri, "Efficient Models for Model Checking Message-Passing Distributed Protocols", Proc. of Formal Techniques for Networked and Distributed Systems (FORTE), 2010.

9. Dan Dobre, Matthias Majuntke, Marco Serafini and Neeraj Suri, "HP: Hybrid Paxos for WANs", Proc. European Dependable Computing Conference (EDCC), 2010.

10. Matthias Majuntke, Dan Dobre, Marco Serafini and Neeraj Suri, "Abortable Fork-Linearizable Storage", Proc. of Int'l Conf. on Principles of Distributed Systems (OPODIS), 2009.

11. Peter Bokor, Marco Serafini and Neeraj Suri, “Role-Based Reduction of Fault-Tolerant Distributed Protocols with Language Support”, Proc. of Int’l Conf. on Formal Engineering Methods (ICFEM), 2009.

12. Dan Dobre, Matthias Majuntke, Marco Serafini and Neeraj Suri, “Efficient Robust Storage using Secret Tokens”, Proc. of Int’l Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS), 2009.

13. Peter Bokor, Marco Serafini, Helmut Veith and Neeraj Suri, “Efficient Model Checking of Fault-tolerant Distributed Protocols Using Symmetry Reduction (Brief Announcement)”, Proc. Int’l Symp. on Distributed Computing (DISC), 2009.

14. Kohei Sakurai, Masahiro Matsubara, Marco Serafini and Neeraj Suri, “Dependable and Cost-Effective Architecture for X-by-Wire Systems with Membership Middleware”, Proc. of FISITA World Automotive Congress, 2008.

15. Peter Bokor, Marco Serafini, Aron Sisak, Andras Pataricza and Neeraj Suri, “Sustaining Property Verification of Synchronous Dependable Protocols Over Implementation”, Proc. of the IEEE Int’l Symp. on High Assurance Systems Engineering (HASE), 2007.