FT-PPTC: An Efficient and Fault-Tolerant Commit Protocol for Mobile Environments*

Brahim Ayari, Abdelmajid Khelil, Neeraj Suri
DEEDS Group
Technische Universität Darmstadt, Germany
{brahim, khelil, suri}@informatik.tu-darmstadt.de

Abstract

Transactions are required not only for wired networks but also for the emerging wireless environments where mobile and fixed hosts participate side by side in the execution of the transaction. This heterogenous environment is characterized by constraints in mobile host capabilities, network connectivity and also an increasing number of possible failure modes. Classical atomic commit protocols used in wired networks are therefore not directly suitable for this heterogenous environment. Furthermore, the few commit protocols designed for mobile transactions either consider mobile hosts only as initiators though not as active participants, or show a high resource blocking time.

We present the Fault-Tolerant Pre-Phase Transaction Commit (FT-PPTC) protocol for mobile environments. FT-PPTC decouples the commit of mobile participants from that of fixed participants. Consequently, the commit set can be reduced to a set of entities in the fixed network. Thus, the commit can easily be supported by any traditional atomic commit protocol, such as the established 2PC protocol. We integrate fault-tolerance as a key feature of FT-PPTC. Performance evaluations confirm the efficiency, scalability and low resource blocking time of our approach.

1 Introduction

The pervasiveness and functionality of portable devices, equipped with wireless network interfaces, are continuously increasing given the rapid progress in wireless technologies. These mobile devices also interact with fixed devices in realizing conventional applications such as e-mail, and also in enabling new applications such as mobile commerce (m-commerce), mobile inventory etc.

*This research is supported in part by Microsoft Research, FP6 IP DECOS and NoE ReSIST

For distributed systems, and especially distributed databases, a transaction is a set of operations that fulfills the following condition: Either all operations are permanently performed or none of them is visible to other operations. This all-or-nothing feature is known as the atomicity property. Commit protocols ensure atomicity and thus constitute a key issue in the execution of transactions. A transaction forms a logical unit of work, such as transfer of money from one bank account to another. Obviously, transactions are also required for many mobile applications such as m-commerce where data consistency defines the applications. Transactions may also involve multiple mobile devices, besides fixed ones, as full participants such as in mobile auctions and mobile inventory.

Mobile environments are characterized by a variety of constraints such as *(a)* the scarcity of processing and energy resources of mobile devices, and *(b)* the continuously varying properties of the wireless channel, which also leads to network disconnections. These constraints make commit protocols that are designed for fixed networks, such as the traditional two-phase commit (2PC) protocol [1], unsuitable for mobile environments. Therefore, a number of commit protocols, such as UCM [2], TCOT [3] and M-2PC [4] have been developed to address these constraints. Unfortunately, existing approaches either only consider mobile hosts as initiators and not as full participants [3], or work only under strong assumptions, such as the homogeneity of database systems [2] or the simultaneous connectivity of all mobile participants at the initiation of the transaction [4]. In addition, the existing protocols typically consider only a small subset of failures in the mobile environment. These drawbacks limit the applicability of existing approaches and pose new challenges for the design of efficient and fault-tolerant commit protocols for heterogeneous mobile environments.

The main contribution of this paper is an efficient, scalable and fault-tolerant atomic transaction commit protocol called *Fault-Tolerant Pre-Phase Transaction Commit (FT-PPTC)*. The key idea is to decouple the commit of mobile hosts from that of fixed hosts. Consequently, we split the

execution of the transaction into two phases: *(1)* the mobile data gathering phase called *pre-commit phase* collecting "sufficient" information from the mobile hosts to provide progress, and *(2)* the core 2PC phase, which involves only fixed hosts for commit action. We also develop a comprehensive fault model for the mobile environment and demonstrate the resilience of the proposed FT-PPTC protocol. During the first phase no resources are blocked on fixed hosts because only mobile hosts are involved. Only if the first phase ends successfully the second phase can start involving fixed hosts. It is important to mention that FT-PPTC does not consider disconnections as exceptional but rather as part of the normal operation of the considered system.

The paper is organized as follows. Related work is discussed in Section 2. In Section 3, we present the different models relevant to this work, including the system model, the transaction model and the associated fault model. The proposed FT-PPTC protocol is described in Section 4. The performance evaluation of our protocol, as well as its comparison to related protocols, is presented in Section 5. Section 6 concludes the paper and outlines our future work.

2 Related Work

Mobile transactions have been the focus of extensive ongoing research [5, 6, 7, 8], and some recent commit protocols have been proposed in the literature [2, 3, 4].

Unilateral Commit for Mobile (*UCM*) [2] supports disconnections and off-line executions on mobile devices. UCM is a one-phase protocol where the voting phase of the 2PC [1] is eliminated. The coordinator acts as a "dictator" imposing its decision on all participants. UCM guarantees atomicity. However due to its strict assumptions (strict two-phase locking [9] required for all participants), the data accessed by uncommitted local transactions remain locked until the whole transaction is committed or aborted. In addition, the UCM assumes homogeneous database systems that is often not viable in mobile environments.

Transaction Commit On Timeout (*TCOT*) [3] uses timeouts to provide a non-blocking protocol that limits the amount of communication between the participants in the execution of the protocol. Instead of exchanging messages to reach a Commit or Abort decision, the coordinator waits for timeouts to expire. TCOT provides only semantic atomicity as defined in [10], which is weaker than the desired strict atomicity [11] for transactions. This limits the applicability of TCOT. Furthermore, TCOT does not consider mobile hosts as active participants in the execution of transactions.

Mobile two Phase Commit (*M-2PC*) [4] considers mobile hosts as active participants. A mobile participant executes its fragments and delegates the commitment to its agent on a fixed host. Unfortunately, M-2PC assumes that all mobile participants are connected at the transaction initiation and that network disconnections are allowed only after the mobile host delegates its commitment duties.

A common drawback of these protocols is that they address only a small subset of the numerous failure types that may occur in the mobile environment. Furthermore, these protocols are not robust to coordinator failures. In our work, we propose a comprehensive fault model and design our protocol to tolerate faults resulting in enhanced transaction resilience compared to existing protocols.

3 System and Fault Models

We first present a model of the mobile environment. Consequently, we elaborate the system model, the transaction and fault models. The fault model describes the fault types that the FT-PPTC protocol is designed to handle.

3.1 Model of the Mobile Environment

We consider a mobile distributed environment consisting of a set of mobile hosts (MH) and a set of fixed hosts (FH). The architecture of the considered environment is based on the so-called Personal Communication System (PCS), where the MHs intermittently connect to the wired network through Mobile Support Stations (MSS) via wireless channels (Figure 1). MHs can communicate with each other or with fixed entities using only the services provided by the MSSs. We refer to the set of MHs as $M = \{MH_1, \ldots, MH_m\}$, and that of FHs as $S = \{FH_1, \ldots, FH_s\}$, where m and s are the number of MHs and FHs respectively.

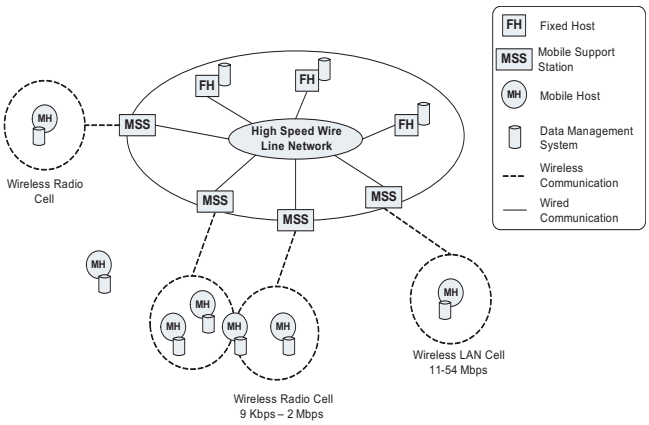

Figure 1. Architecture of Mobile Environment

We assume that each MH has a Mobile Database Server (MDBS) installed on it, and that a Database Server (DBS) is attached to each FH. Database servers are needed on both

fixed and mobile hosts to support basic transaction operations such as read, write, commit and abort.

3.2 System Model

We consider applications, which run on either mobile or fixed hosts and access data located on both *mobile* and *fixed* hosts. A transaction can originate from any host in $M \cup S$, and the participants in its execution can be any set $P \subseteq M \cup S$. However, most of mobile transactions involve some FHs as participants.

The hosts in the considered mobile environment may entail different hardware and different software, including their *database management systems*. Thus, we are dealing with *heterogenous* mobile databases. MHs are also heterogenous and can range from mobile phones with restricted storage and processing capabilities to laptops with considerably higher capabilities.

We consider all distributed database system components ($\in M \cup S$) to be *autonomous*. Autonomy means that the components of the system are able to perform their tasks independently from each other. With respect to the execution of transactions, this requires that every component must take the decision to commit or abort the transaction independently from other components in the network. Components are also able to decide which information to share with the global system and how to manage their own data. The data of the MH is replicated on a fixed backup database server.

We assume the existence of a *coordinator (CO)*, which is responsible for coordinating the execution of the corresponding transaction. For different transactions, different nodes may play the CO role. The CO is responsible for storing information concerning the state of the transaction execution. Based on the information collected from the participants of the transaction, the CO takes the decision to commit or abort the transaction and informs all participants about its decision.

The MH is not considered to have a physical stable storage, since it is subject to loss, damage and random disconnections. For this reason the MH is not desired to take on the CO role. We assume only FHs to have a stable storage. Thus, the CO role should be performed by one or more FHs. If a MSS has similar capabilities to that of a FH, the CO role can also be performed by a MSS. The CO maintains information about the connectivity of the MHs participating in the execution of the transaction (i.e., whether they are connected to the network or not).

We do not place any restrictions on the storage capabilities of FHs. Further we assume that all DBSs attached to the FHs support the $prepare$ operation of the 2PC protocol as a basic operation.

3.3 Transaction Model

Users issue transactions from MHs. After processing a transaction, the system provides the result of the transaction on the user's MH. The transaction may be entirely executable at the user's MH. In most of cases, however, the transaction has to be fragmented and distributed among a set of nodes $P \subseteq M \cup S$. We refer to a distributed transaction where at least one MH participates in its execution as a *Mobile Transaction (MT)*. Similar to the concept of transaction formalization presented in [12], we formally define the MT T_i as a triple $< F_i, L_i, FLM_i >$, where $F_i = \{e_{i1}, e_{i2}, \ldots, e_{in}\}$ is a set of n "execution fragments" [13, 14], $L_i = \{l_{i1}, l_{i2}, \ldots, l_{ik}\}$ is a set of k locations in $M \cup S$ ($k \leq m + s$), and $FLM_i = \{flm_{i1}, flm_{i2}, \ldots, flm_{in}\}$ is a set of fragment-location mappings (flm's), where $\forall j, flm_{ij}(e_{ij}) = l_{ij}, 1 \leq j \leq k$. Although the execution fragments of T_i are semantically related, each one of them can commit independently, given the autonomy of their corresponding locations, leading to the commit of T_i.

We refer to the MH, where T_i is initiated, as *Home MH (H-MH)*. The *commit set* consists of all FHs and MHs participating in execution and commit of T_i including H-MH. FHs and MHs in the commit set are called *participant FHs (Part-FH)* and *participant MHs (Part-MH)* respectively.

The database system installed on MHs provides *backup* facilities to assist with the *recovery* of the database. To achieve a 100% recovery, all the operations completed on the MH need to be stored on a stable and reliable storage on a FH. Thus, the MHs are able to take part in the execution of transactions in the considered environment independently of other system components.

3.4 Fault Model

Designing a fault-tolerant transaction commit protocol essentially requires the identification of constraints and failure modes that can occur in the considered environment. The following sections enumerate these aspects.

3.4.1 Constraints

The PCS environment is constrained mainly by the characteristics of *MHs* and *wireless links*. *MHs* intuitively possess less computational resources such as processor speed and storage capacity than the FHs. This increases the time MHs need to execute transactions or may even lead to execution failure. Furthermore, MHs are highly vulnerable to physical loss or damage, and may run in different energy modes or be put-off to save energy. Therefore, MHs naturally show frequent and random network disconnections.

The effective bandwidth available for the MH over a *wireless link* is highly dynamic. This depends on the wireless technology, access coverage, and number of MHs that have to share the medium. These characteristics lead to an unreliable and intermittent network connectivity of MHs.

The limitations and characteristics listed above outline the variation of constraints for the PCS environment being different from those in fixed environments. These constraints also make it harder to design appropriate and efficient commit protocols. A protocol that aborts the transaction, each time the MH disconnects from the network, is not suitable for mobile environments, since frequent disconnections are not exceptional but are rather part of the normal mode of operation. Therefore, disconnections need to be explicitly tolerated by the protocol.

3.4.2 Failure Modes

We now outline the considered failure modes classified into primary classes of communication and node failures.

Communication Failures: These constitute the most frequent failures in the mobile environment. We distinguish between three kinds of communication failures:

- *Message loss:* Messages exchanged between the MH and the MSS are highly vulnerable to loss due to the high transmission error rate of wireless links and to network congestion.
- *Communication delay:* Higher end-to-end delay is mainly caused by network congestion.
- *Network disconnection:* While moving, the MH can enter a geographical area out of coverage of any MSS so that it loses its connection to the network.

Node Failures: We distinguish between MH, FH and CO failures. We separate CO failures from FH failures given the central role CO plays in commit protocols. For MHs, we classify the failures into *transient* and *permanent* failures.

- *Transient MH failures:* These occur from either software or hardware faults and usually disappear if the MH is restarted. A further common cause of transient failures is the lack of battery power to sustain operation of the mobile device. Transient failures are the most probable failures of MHs in the PCS environment. In the case of a transient MH failure, the content of the volatile storage of the MH and consequently the state of its recent computations is lost.
- *Permanent MH failures:* These are irreparable failures such as the loss or damage of the MH itself or its non-volatile storage, where the data and logs are stored. Consequently, all the data stored in the MH is lost.
- *CO failures:* We assume a crash-recovery model, i.e., if the CO crashes it stops receiving, sending and processing messages until it recovers after a finite amount of time. Volatile storage of the CO is checkpointed periodically to stable storage and the CO logs its computations and received/sent messages between two checkpoints. Once a backup is done the log is deleted and a fresh logging process is initiated.
- *FH failures:* We assume a crash-recovery model but limit its details as our focus is on failures that are specific to the described mobile environment.

4 The FT-PPTC Protocol

We introduce FT-PPTC, a novel atomic commit protocol for mobile transactions. First, we present a short overview of the approach. Next, the failure-free operation of FT-PPTC is discussed and its fault-tolerance aspects are outlined. Finally, the FT-PPTC's correctness is proved.

4.1 The Proposed Approach: Overview

As key drivers, the FT-PPTC commit protocol has to ensure the atomicity property. It should efficiently minimize the number of transaction aborts by tolerating the failures described in the fault model. High efficiency is reflected by a low message complexity, especially for wireless messages. Since some FHs participate in most of the transactions, FT-PPTC should reduce the blocking time of resources at the Part-FHs.

As MHs may need an arbitrary long time to execute their fragments, and as very few assumptions can be made regarding the connection intervals of MHs, resources of fixed participants may potentially be blocked for an undefined period of time. Therefore, we suggest to *decouple* the commit of mobile participants from that of fixed participants. We split transaction execution into *two phases*. The first phase, called the *pre-commit phase* (Figure 2), collects "sufficient" information from mobile participants and reduces the commit set to a set of entities in the fixed network. In the second phase the commit involves only FHs and thus can be completed by any atomic commit protocol for wired networks, such as the traditional 2PC protocol [1]. We refer to the second phase as the *core 2PC phase*.

To allow for this decoupling, we assign a *MH Agent (MH-Ag)* to each Part-MH. The MH-Ag is representing the Part-MH in the fixed network. The MH-Ag is responsible for storing all the information related to the state of all MTs involving the MH. The MH-Ag is also responsible for executing the 2PC protocol on the behalf of its corresponding Part-MH. The MH-Agent can be implemented by any FH. The CO itself is the MH-Ag of the H-MH.

Intuitively, this decoupling reduces the blocking time of the resources at the fixed devices. It also simplifies the han-

dling of the different types of failures that arise from the mobility of nodes as described in the fault model.

4.2 Pre-Commit Phase

The pre-commit phase only involves Part-MHs. The MH-Ags act as intermediators between Part-MHs and the CO. Similar to [3], we exploit a timeout-based concept to reach a provisional Commit decision at the end of the pre-commit phase. Each Part-MH computes an *execution timeout (E_t)*, an upper bound for the time to complete the execution of the transaction fragment, and a *shipping timeout (S_t)*, an upper bound for the time to compose updates and to send them to the CO. Both timeouts have to account for the constraints of the MH and the wireless link. These timeouts can be extended if needed. Each mobile participant sends both timeout values E_t and S_t to the CO via its MH-Ag.

The CO waits for the expiration of the timeouts of Part-MHs and collects their votes along with the data logs of the H-MH in case of successful execution. The data logs contain the list of all updates made by the MT. The data logs of other Part-MHs (in case of successful execution) are stored by their corresponding MH-Ags. The MH-Ag should be able to propagate the updates made by the Part-MH (in case this has a permanent failure) to its corresponding backup database server using the logs. The CO finalizes the pre-commit phase by a provisional Commit decision or a final Abort decision. The CO decides to proceed to the second phase of FT-PPTC, only if it receives the updates from the H-MH and a "Yes" vote from all MH-Ags (representing the rest of Part-MHs) within the specified time-limit. The transaction is aborted as soon as one Part-MH aborts the transaction or the timeout expires at the CO without receiving either the updates of the H-MH or a "Yes" vote from one of the MH-Ags.

4.3 Core 2PC Phase

As a result of the pre-commit phase, the Part-MHs delegate their corresponding MH-Ags to execute the 2PC protocol on their behalf. The second phase of the protocol begins, when the CO sends the execution fragments of Part-FHs to their corresponding FHs and the 2PC protocol is executed to collect their votes. If all Part-FHs vote for committing the MT, the CO decides to commit it, otherwise it decides to abort. We assume that the 2PC protocol used for collecting the votes from the Part-FHs is modified in such a way that it is non-blocking. This can be achieved using, for example, timeouts to detect message loss.

4.4 Operation of the FT-PPTC Protocol

Figure 2 illustrates the failure-free execution of the FT-PPTC protocol. The activities of each participant are outlined below. Specifically, we refer the activities of H-MH, CO, MH-Ag and Part-MH to the details in Algorithm 1, Algorithm 2, Algorithm 3 and Algorithm 4 respectively.

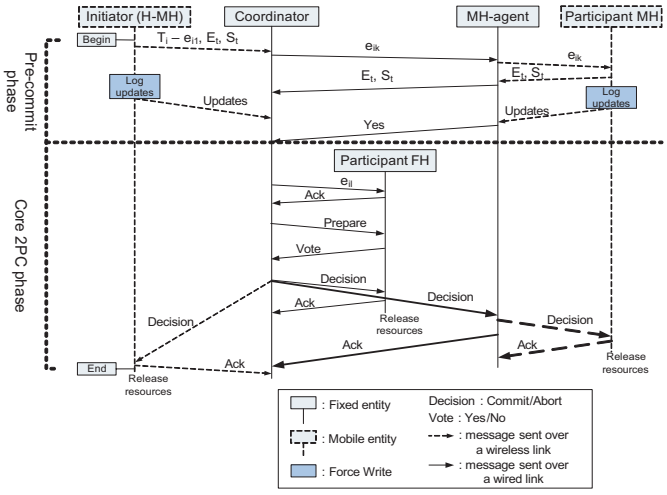

Figure 2. Failure-free execution of the MT using the FT-PPTC protocol

Activities of the H-MH

The H-MH initiates the mobile transaction T_i, extracts its execution fragment e_{i1}, computes its E_t and S_t and sends the rest of the MT $T_i - e_{i1}$ along with E_t and S_t to the CO (Algorithm 1). The H-MH begins processing of e_{i1}. Whenever the H-MH needs to extend its E_t or S_t, it sends a message to the CO with the new timeout value which needs to be extended. If the H-MH decides to abort the MT before the expiration of its timeout $E_t + S_t$, then it sends an Abort message to the CO. If the H-MH completes successfully the execution of its fragment, it composes its updates (the updates are basically the logs of the MH), writes them on its non-volatile storage (the writing of the updates has to be finished before sending them - "force write") and sends them to the CO. The H-MH acknowledges the reception of the decision by sending an Ack message to the CO.

Activities of the CO

Upon receiving $T_i - e_{i1}$ from the H-MH, the CO creates a *Token* for T_i, which includes one entry for each e_{ik} and contains the identity of the CO and the commit set (Algorithm 2). Each entry includes the state of processing of e_{ik}. We distinguish between the following states: (a) *idle*, (b) *active*, (c) *pre-committed* (only for MHs), (d) *committed* and (e) *aborted*. The state of e_{i1} is set to active and the state of the rest of the execution fragments is set to idle. The entries for the execution fragments of the MHs additionally include their corresponding E_t and S_t. Thereafter, it ex-

Algorithm 1: H-MH's Algorithm

```
1  Initialize T_i;
2  Extract its execution fragment e_i1;
3  Compute E_t(H-MH) and S_t(H-MH);
4  send T_i − e_i1, E_t(H-MH) and S_t(H-MH) to CO;
5  while processing e_i1 do
6      if E_t(H-MH) needs to be extended then
7          Extend E_t(H-MH);
8          send new value of E_t(H-MH) to CO;
9      end
10 end
11 if H-MH decides to abort T_i then
12     write Abort record in the local log;
13     send Abort message to CO;
14     return;
15 else   // H-MH decides to commit T_i
16     while composing updates do
17         if S_t(H-MH) needs to be extended then
18             Extend S_t(H-MH);
19             send new value of S_t(H-MH) to CO;
20         end
21     end
22     force write updates to the local log;
23     send updates to CO;
24     wait for decision message from CO
25     if decision message is Commit then
26         commit T_i;
27         write Commit record in the local log;
28         send Ack message to CO;
29         return;
30     else   // decision message is Abort
31         abort T_i;
32         write Abort record in the local log;
33         send Ack message to CO;
34         return;
35     end
36 end
```

Algorithm 2: Coordinator's Algorithm

```
1  wait for T_i − e_i1, E_t(H-MH) and S_t(H-MH) from H-MH;
2  create a Token for T_i;
3  extract execution fragments of the Part-MHs and sends them to
   their corresponding MH-Agents;
4  initialize all the timeouts of the Part-MHs with 0;
5  let P_mp = {MH_1, ..., MH_mp} the set of all the Part-MHs;
6  tm_i ←
   max(E_t(MH_1) + S_t(MH_1), ..., E_t(MH_mp) + S_t(MH_mp));
7  while waiting for tm_i to expire do
8      if value of E_t or S_t (initial or extended value) of one of the
       MHs in P_mp is received then
9          recompute tm_i;
10         update the Token of T_i with the received value(s);
11     end
12     if Abort message is received from one of the MHs in P_mp
       then
13         write Abort in T_i's Token;
14         send Abort to all members of P_mp;
15         return;
16     end
17 end
18 if updates are received from H-MH and a Yes vote from each
   MH-Ag then
19     write all received updates;
20     start a 2PC protocol to collect the votes from all Part-FHs;
21     if all votes were Yes then
22         write Commit in T_i's Token;
23         send Commit message to all members of the commit
           set;
24         return;
25     else   // at least one of the votes is No
26         write Abort in T_i's Token;
27         send Abort to all members of the commit set;
28         return;
29     end
30 else   // at least the updates of one of the
   // MHs in P_mp are not received
31     write Abort in T_i's Token;
32     send Abort to all members of the commit set;
33     return;
34 end
```

tracts the execution fragments of the Part-MHs and sends them to their corresponding MH-Ags. After receiving E_t and S_t from any Part-MH, the CO sets the state of its corresponding fragment to active. If a new E_t or S_t (extended values) is received either from H-MH or from a Part-MH, then the CO updates the Token. The CO waits for the expiration of the maximum value of $E_t + S_t$ it has received. If it receives the updates from the H-MH and a "Yes" vote from each MH-Ag within this time, it stores them and sets the state of their corresponding fragments to pre-committed. After that the execution fragments of the Part-FHs are sent to their corresponding FHs along with the vote request (prepare message). If the CO receives an Abort message from any of the Part-MHs before the expiration of the timeout or if it doesn't receive the updates from at least one MH within this timeout, it sets the state of all fragments to aborted and sends an Abort message to the rest of the Part-MHs and the whole transaction is aborted. After sending the execution fragments of the Part-FHs, the CO starts a 2PC protocol session to collect the votes from them. If the CO receives a "Yes" vote from all the Part-FHs, it decides to commit the transaction and sends Commit decision to all the participants. If it receives at least one "No" vote (or no reply) it decides to abort the transaction and sends Abort decision to all the participants.

Activities of the MH-Ag

Upon receiving the execution fragment of its corresponding Part-MH, the MH-Ag creates a Token for this fragment and stores it in stable storage (Algorithm 3). It then forwards the fragment to the Part-MH. After receiving E_t and S_t from the Part-MH, the MH-Ag adds them to the corresponding Token and forwards this information to the CO. After receiving the updates from the Part-MH, the MH-Ag stores them in the corresponding Token and informs the CO

by sending a "Yes" vote to it. Upon receiving the decision from the CO, the MH-Ag sends an acknowledgment to the CO and stores the decision in the corresponding Token and forwards it to the Part-MH as soon as it reconnects to the network. Any information exchange between the Part-MH and CO is stored in its corresponding Token in the stable storage of the MH-Ag before forwarding it. The MH-Ag is not an active participant in the execution of the MT, since it does not have to know any information about the application and does not need to process any part of the MT.

Algorithm 3: MH-Ag's Algorithm

```
1  wait for receiving execution fragment e_ik of the corresponding
   Part-MH from CO;
2  create a Token for e_ik;
3  for any received message do
4      if message is sent by the CO then
5          update the Token with the received message;
6          send the received message to the corresponding
           Part-MH;
7      else if message contains the updates of the corresponding
       Part-MH then
8          update the Token with the received updates;
9          send Yes vote to CO;
10     else
11         update the Token with the received message;
12         send the received message to CO;
13     end
14 end
```

Activities of the Part-MH

Upon receiving its execution fragment, the Part-MH calculates its E_t and S_t, and sends them back to its MH-Ag (Algorithm 4). It then starts processing its execution fragment. Whenever the Part-MH needs to extend its E_t or S_t, it informs the CO (through its MH-Ag) about the new value. If the Part-MH decides to abort the MT before the expiration of its timeout, it sends an Abort message to its MH-Ag. If the Part-MH successfully completes the execution of its fragment, it composes its updates and sends them to its MH-Ag. Upon the reception of the decision, the Part-MH acknowledges it by sending an Ack message to its MH-Ag.

Algorithm 4: Part-MH's Algorithm

```
1  wait for receiving the corresponding execution fragment;
2  Compute E_t(Par-MH) and S_t(Par-MH);
3  send E_t(Par-MH) and S_t(Par-MH) to MH-Ag;
   // continue with step 5 to step 36 of
      Algorithm 1 substituting CO with MH-Ag
```

Activities of the Part-FH

Part-FHs behave according to the 2PC protocol [1], i.e., a Part-FH executes its fragment, waits for the prepare message, sends its vote and waits for the decision from the CO.

4.5 Resilience of the FT-PPTC Protocol

We now outline the fault-tolerance aspects of FT-PPTC for its handling of communication and node failures.

4.5.1 Handling Node Failures

Fixed Participants Failures: FH failures are handled directly as in the 2PC protocol [1].

Mobile Participants Failures: If the Part-MH has a transient failure before finishing the execution of its fragment and it recovers from it before its timeout expires, then it can extend this timeout by sending a message to the CO (via its MH-Ag) with the new values of E_t and S_t. All the information about the MT can be found in the non-volatile storage of the MH since only the content of the volatile storage get lost if the MH has a transient failure. If the MH recovers from the failure after the expiration of its timeout or if it has a permanent failure, then the transaction will be aborted since the CO will not receive the updates in time.

Coordinator Failures: If the CO crashes before receiving the timeout values from a certain Part-MH, it asks the MH-Ag of this participant for this information upon recovery. The MH-Ag responds with the timeout values only if the values are available otherwise it asks the Part-MH for this information before responding. The CO initializes the timeout values with 0 and updates them upon the reception of these timeout values. If this information does not reach the CO because they are lost or the Part-MH crashes before sending them, these values remain set to 0. If the CO crashes before the expiration of the timeout $E_t + S_t$, it retrieves the Token of the MT from its logs or backups after recovering. In this case the CO asks the Part-MHs if anyone of them has extended its timeouts or sent its updates (H-MH) or vote (MH-Ag) while the CO was down. The updates sent by the Part-MHs are stored by their corresponding MH-Ags and a "Yes" vote is sent to CO after it recovers. Since the CO has a stable storage it can read the status of the execution of the MT and the state of each execution fragment stored there after recovering from failures and continues to process the MT. The CO can decide about the information it might have lost to request this information. As example, if the CO after recovering reads an idle state of an execution fragment of a Part-MH in the Token of the MT, it sends a message to the corresponding MH-Ag asking it for the values of the timeouts. Similarly, if the CO after recovering finds in the Token any execution fragment having the state aborted, then it sends an Abort message to all the participants.

4.5.2 Handling Communication Failures

Longer delays of messages can be handled by tuning the variable S_t. The value of S_t should take into account the current estimated values of communication delays to decrease the number of aborted MTs. If the available bandwidth is low, the Part-MH sends a message to the CO to extend the value of its S_t.

If the updates sent by a Part-MH are lost or delayed due to network disconnection or congestion, the CO will decide to abort the transaction after the expiration of the timeouts.

4.6 Proof of Correctness

According to [9], an atomic commit protocol has to satisfy the following five *atomicity conditions (AC)*:

- *AC1:* All processes that reach a decision reach the same one.
- *AC2:* A process cannot reverse its decision after it has reached one.
- *AC3:* The Commit decision can only be reached if *all* processes voted "Yes".
- *AC4:* If no failure occurs and all processes voted "Yes", then the final decision should be commit.
- *AC5:* Consider any execution containing only failures that the protocol is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long time, then all processes will eventually reach a decision.

To prove the correctness of the proposed FT-PPTC protocol, we have to prove that it satisfies the five conditions listed above. It follows directly from the specification of the protocol that the FT-PPTC protocol satisfies conditions AC1, AC2 and AC4. Therefore, we need only to prove that it satisfies AC3 and AC5.

[AC3 Correctness]: We assume that the CO decides to commit the transaction when at least one of the participants has not decided yet. If this participant is a MH then the CO sends the rest of the transaction to the FHs before receiving a "Yes" vote from its MH-Ag or before receiving the updates if this MH is the H-MH. Obviously this is in contradiction with our protocol specification. If this participant is a FH, then the 2PC protocol decides to commit the transaction before receiving all the votes from the Part-FHs, which again contradicts the specification of the 2PC protocol. In the case that at least one of the participants decides to abort the transaction, the CO can not decide to commit the whole transaction because this decision will violate the protocol specification. Hence, the Commit decision can only be reached if all processes voted "*Yes*", i.e., decided to commit the transaction. □

[AC5 Correctness]: We consider any execution containing the failures listed in the fault-model detailed in Section 3.4. For this proof we need to consider two aspects. The first aspect is, whether the protocol can block at any time so that the participants will not be able to take a decision anymore. Since our protocol is based on timeout for coordinating the execution of the fragments of Part-MHs, the CO can not block waiting for messages from these participants. The participants also can not block waiting for a message from the CO (which is the decision) since they are required to acknowledge this message and thus the CO is able to detect a message loss and re-sends the message. The second aspect considers whether the participants are able to reach any decision after recovering or not. This aspect is handled in the failure scenarios in Section 4.5. We note that keeping the state of the execution of MT on a stable storage allows the continuation of the execution after recovery and eventually reaching a decision. □

5 Performance Evaluation

To evaluate the efficiency of our approach, we first compare it to the commit protocols TCOT [3], UCM [2] and M-2PC [4], regarding their failure-tolerance. Next, we study the message complexity in the failure-free case. Finally, we introduce a simulation model and investigate the performance of the FT-PPTC protocol in the failure-free case with respect to the throughput and resource blocking time.

5.1 Comparison to Related Work

We now compare the FT-PPTC protocol to the TCOT, UCM and M-2PC protocols regarding the failures considered in each protocol and the message complexity.

5.1.1 Failures Discussed in Different Protocols

In Table 1, we compare the FT-PPTC protocol to the M2-PC, TCOT and UCM protocols spanning the failures detailed in our fault model. FT-PPTC is the only protocol that can handle CO failures. The comprehensive handling of communication and node failures distinguishes FT-PPTC.

5.1.2 Message Complexity for the Failure-Free Case

We denote by e the number of timeout extensions of Part-MHs and by e_{all} the number of all timeout extensions. m_p represents the number of Part-MHs and s_p the number of Part-FHs.

We adopt the message complexity of TCOT, M-2PC and UCM from [4, 2]. We refer to Figure 2 to compute the message complexity of FT-PPTC. It follows that each Part-MH sends two messages to the corresponding MH-Ag and receives one message from it starting from the point in time, where the updates are shipped. Whenever a Part-MH needs

Table 1. Coverage of failures

Protocol	Transient MH-Failure	Permanent MH-Failure	Coordinator Failure	Message loss	Delay of messages	Network disconnections
FT-PPTC	x	x	x	x	x	x
M-2PC	x			x	x	x
TCOT	x	x		x	x	x
UCM	x			x	x	x

Table 2. Comparison of the protocols in failure-free case

Protocol	Atomicity	Phases	Wireless Message Complexity	Overall Message Complexity
FT-PPTC	strict	2	$(2+1)*m_p+e$	$(3m_p+e)+(4s_p+2m_p+e)$
M-2PC	strict	2	$(2+2)*m_p-1$	$(4m_p-1)+4s_p$
TCOT	semantic	1	$2*m_p-1+e$	$(2m_p-1)+(2s_p)+e_{all}$
UCM	strict	1	$(1+1)*m_p$	$2m_p+2s_p$

to extend its timeouts (E_t and S_t) it sends an extra message to the MH-Ag.

Table 2 details the efficiency of FT-PPTC compared to other protocols, while providing strict atomicity. We emphasize that FT-PPTC's efficiency is comparable to classical protocols, even though it allows for fully mobile participants.

5.2 Simulation

We now present our simulation model and our preliminary results for the failure-free case. We compare the performance of the FT-PPTC protocol to that of M-2PC [4] and the conventional 2PC.

5.2.1 Simulation Model

For our simulation studies, we have used SimJava [15], a discrete event-based simulator implemented in Java. Table 3 summarizes our simulation parameters.

Table 3. Simulation settings

Parameter	Value
#Part-FHs	4
#MHs	$\in$ [1,25]
Execution time of one fragment (MH)	5 ms
Execution time of one fragment (FH)	2 ms
Transmission delay over wireless link	10 ms
Transmission delay over wired link	5 ms
#Fragments per MT	n = 5

We generate transactions as follows. We assume all transactions are of similar length and are composed of n fragments. We let each MH initiate one transaction at the beginning of the simulation. Therefore, the number of initiated transactions is identical to the number of MHs.

5.2.2 Simulation Results

In our simulation-based performance analysis, we focus on two performance metrics: Throughput and resource blocking time. We define the *throughput* as the number of successfully committed MT per time unit, and the *resource blocking time* as the time interval, where the resources at the *fixed participants* remain locked.

Throughput: We first compare the throughput of FT-PPTC to M-2PC and standard 2PC. Figure 3 shows the throughput over the number of transactions. We observe that M-2PC shows a slightly higher throughput than FT-PPTC, due to the fact that FT-PPTC decouples the execution of fragments of Part-MHs and Part-FHs, whereas these fragments are executed in parallel in M-2PC. Compared to 2PC, FT-PPTC shows a higher throughput. This occurs as 2PC needs one more wireless message, which increases the commit time.

Overall FT-PPTC shows a stable performance behavior that is similar to the behavior of the traditional 2PC protocol for fixed devices. This is significant given that the effect of mobile hosts is shown to be minimal for the commit operations. It also validates the effectiveness of our split two-phase approach, where the impact of the decoupling in FT-PPTC on the performance is minimal.

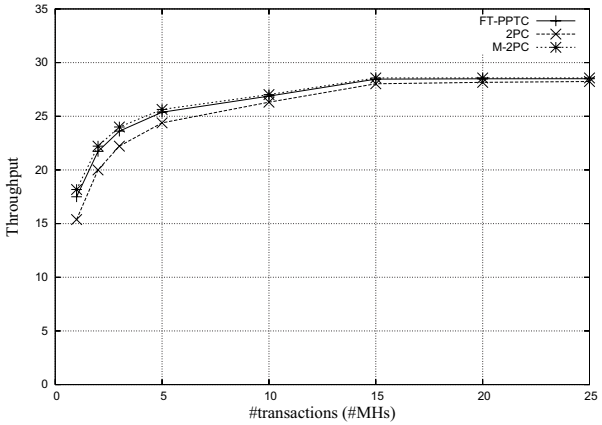

Figure 3. Throughput

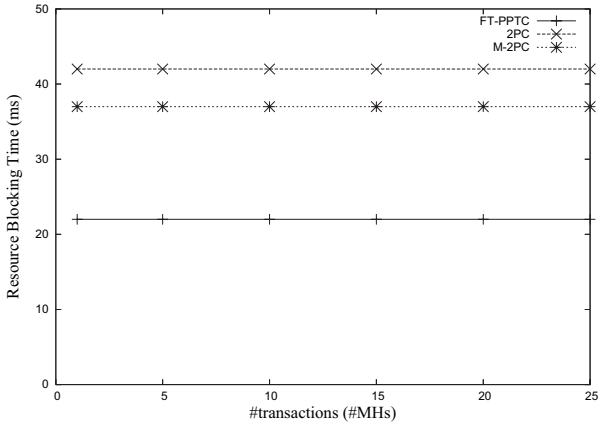

Figure 4. Resource blocking time

Resource Blocking Time: In the following we compare the resource blocking time of the three protocols. Figure 4 depicts the blocking time over the number of transactions. FT-PPTC shows a significantly lower blocking time of the resources due to the decoupling of the commit of Part-MHs from that of Part-FHs. This decoupling makes the resource blocking time in FT-PPTC *only* dependent on the time needed by Part-FHs to execute their corresponding fragments, which is considerably shorter than the time needed by Part-MHs. This also demonstrates that FT-PPTC is scalable regarding the number of MTs, since the resource blocking time remains constant over the number of MTs.

Furthermore, we emphasize that the resource blocking time of FT-PPTC is independent from the number of mobile participants. This enhances the scalability of our approach.

6 Conclusion and Future Work

In this paper, we have presented the FT-PPTC approach, a fault-tolerant atomic commit protocol for mobile transactions. FT-PPTC decouples the commit of mobile participants from that of fixed participants. This approach is shown to reduce the blocking time of resources on the fixed part of the network and allows a high resilience to both communication and node failures. Specifically, the performance analysis shows that FT-PPTC is efficient and also scalable regarding the number of transactions and the number of mobile participants. We consider the efficiency and scalability, while providing for fault-tolerance, to be highly useful attributes for the mobile environment.

In future work, we plan to extend the fault model and design efficient fault-recovery mechanisms. Our long-term goal is to consider varied communication models, such as ad hoc communication between the mobile devices.

References

[1] Gray, J. *Notes on Data Base Operating Systems*. In Operating Systems, An Advanced Course, pp. 393–481. 1978.

[2] Bobineau, C. et al. *A Unilateral Commit Protocol for Mobile and Disconnected Computing*. In Proc. PDCS. 2000.

[3] Kumar, V. et al. *TCOT-A Timeout-Based Mobile Transaction Commitment Protocol*. IEEE Trans. on Computers, 51(10): pp. 1212–1218, 2002.

[4] Nouali, N. et al. *A Two-Phase Commit Protocol for Mobile Wireless Environment*. In Proc. 16^{th} Australasian Database Conference, pp. 135–143. 2005.

[5] Dunham, M. H. et al. *A Mobile Transaction Model That Captures Both the Data and Movement Behavior*. Mobile Networks and Applications, 2(2): pp. 149–162, 1997.

[6] Chrysanthis, P. K. *Transaction Processing in Mobile Computing Environment*. In IEEE Workshop on Advances in Parallel and Distributed Systems, pp. 77–83. 1993.

[7] Pitoura, E. et al. *Maintaining consistency of data in mobile distributed environments*. In Proc. 15^{th} ICDCS, pp. 404–413. 1995.

[8] Madria, S. K. et al. *A Transaction Model to Improve Data Availability in Mobile Computing*. Distributed Parallel Databases, 10(2): pp. 127–160, 2001.

[9] Bernstein, P. A. et al. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] Garcia-Molina, H. *Using semantic knowledge for transaction processing in a distributed database*. ACM Transactions on Database Systems, 8(2): pp. 186–213, 1983.

[11] Haerder, T. et al. *Principles of transaction-oriented database recovery*. Morgan Kaufmann Publishers Inc., 1994.

[12] Ozsu, M. T. et al. *Principles of distributed Database Systems*. Prentice-Hall, Inc., 1991.

[13] Kumar, V. et al. *Defining Location Data Dependency, Transaction Mobility and Commitment*. TR 98-CSE-1, Southern Methodist Univ., February 1998.

[14] Kumar, V. *A Timeout-Based Mobile Transaction Commitment Protocol*. In Proceedings of the East-European Conference on Advances in Databases and Information Systems, pp. 339–345. 2000.

[15] *SimJava*. http://www.dcs.ed.ac.uk/home/hase/simjava.