Partition Aware Database Replication
A state-update transfer strategy based on PRiDe

Johan Olby
Partition Aware Database Replication
A state-update transfer strategy based on PRiDe

Submitted by Johan Olby to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics.

December 14, 2007

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signature: __________________________________________
Partition Aware Database Replication
A state–update transfer strategy based on PRiDe

Abstract
Distributed real-time databases can be used to support data sharing for applications in wireless ad-hoc networks. In such networks, topology changes frequently and partitions may be unpredictable and last for an unbounded period. In this thesis, the existing database replication protocol PRiDe is extended to handle such long-lasting partitions. The protocol uses optimistic and detached replication to provide predictable response times in unpredictable networks and forward conflict resolution to guarantee progress.

The extension, pPRiDe, combines update and state transfer strategies. Update transfer for intra-partition communication can reduce bandwidth usage and ease conflict resolution. State transfer for inter partition conflicts removes dependency on a common state between partitions prior to the merge to apply update messages on. This makes the resource usage independent of the life span of partitions. This independence comes at the cost of global data stability guarantees and pPRiDe can thus only provide per partition guarantees. The protocol supports application specific conflict resolution routines for both state and update conflicts. A basic simulator for mobile ad-hoc networks has been developed to validate that pPRiDe provides eventual consistency.

pPRiDe shows that a hybrid approach to change propagation strategy can be beneficial in networks where collaboration by data sharing within long lasting partitions and predictable resource usage is necessary. These types of systems already require the conflict management routines necessary for pPRiDe and can benefit from an existing protocol.

In addition to pPRiDe and the simulator this thesis provides a flexible object database suitable for future works and an implementation of PRiDe on top of that database.

Keywords: Data Replication, Distributed Databases, PRiDe, MANET, Optimistic Replication, Real-Time Systems
# Contents

1 Introduction 1

2 Background 3
   2.1 Real-Time Systems 3
   2.2 Mobile Ad-Hoc Wireless Networks 4
   2.3 View-Synchronous Group Communication 4
   2.4 Database Systems 7
   2.5 Replication and Consistency 8
   2.6 PRiDe 11

3 Partitions in Distributed Real-Time Databases 14
   3.1 Aim 14
   3.2 Motivation 14
   3.3 Objectives 15
   3.4 Assumptions 15

4 Method 17
   4.1 Implementation of a database with PRiDe replication 17
      4.1.1 Architecture 18
      4.1.2 Stored Objects 19
   4.2 pPRiDe 19
      4.2.1 Network Partitions 20
      4.2.2 Update propagation in pPRiDe compared to PRiDe 21
   4.3 Simulator 23
      4.3.1 Node Movement 24
      4.3.2 Communication 24
      4.3.3 Database Load 25

5 Implementation 26
   5.1 Communication messages 26
   5.2 Data Structures and Basic Functions 27
   5.3 pPRiDe 29

6 Evaluation 34
   6.1 Proof sketch for pPRiDe 34
   6.2 Validation 38
   6.3 Comparison with existing protocols 40
      6.3.1 PRiDe 41
      6.3.2 Bayou 41
      6.3.3 Bengal 41
7 Conclusions 43

7.1 Summary 43

7.2 Contributions 44

7.2.1 pPRiDe 44

7.2.2 PRiDe implementation 44

7.2.3 Simulation platform 45

7.3 Discussion 45

7.4 Future work 46

7.4.1 Voluntary partitions 46

7.4.2 Communication overhead 46

7.4.3 Storage overhead 47

7.4.4 One way communication 47

7.4.5 Transactions 47

7.4.6 Adding and removing objects 47

7.4.7 "Fuzzy" keys 47

References 49
The advent of low cost mobile computer devices has changed the way we work with computers. The popularity of personal digital assistants (PDA), cell phones, mp3 players, etc. has made mobile computing a commonplace phenomenon. Lately, short distance packet radio systems have been added to devices of this class. This brings a new type of distributed systems, peer-to-peer systems over ad-hoc networks with low powered mobile devices as network nodes. Existing protocols for data sharing must be adapted to handle frequent changes in network topology and unpredictable network partitions since the mobile nodes can move out of reach of each other or power down.

This type of devices can, for example, be used by rescue personnel in emergency response scenarios. The workers use PDAs to access and update a shared view of the situation and the commanders can use it to give instructions. The situation may not allow the workers to delay the rescue operation until they have secured a communication infrastructure. However, an ad-hoc network can provide a possible means of data exchange. A shared whiteboard architecture based on a distributed real-time database can be a suitable architecture for this type of systems (Syberfeldt 2007). The local application updates the data as if the data was stored in a single local database. The network communication and data propagation is handled by the database manager. The unreliability of the ad-hoc network makes it impossible to guarantee that all nodes can be reached to verify and propagate the updates of one node. This may result in multiple divergent replicas of the same object that must be merged when the data can be synchronised between divergent copies.

We present a database replication protocol that extends the PRiDe (Andler et al. 2007) replication protocol with support for arbitrarily long lived partitions while still keeping the real-time characteristics of the protocol. Updates are optimistically applied locally without relying on network communication. The tentative result of local updates is immediately available to consecutive database operations on the local replica. The application can optionally choose to read the possibly older value that is agreed to be stable by the nodes in the current network partition. The result of tentative operations will eventually be agreed to be stable by the nodes in
the partition. The conflict resolution is performed in forward direction where an application provided function resolves a set of conflicting updates into a single non conflicting update. Propagation of updates requires a common start state to result in the same final state. This makes update propagation insufficient when partitions with replicas of the same object, but derived from different initial states, merge. To resolve this, a strategy for how to derive and agree upon a common state is needed. For this purpose, the protocol propagates the different states between the members and a user provided state resolution policy is used to derive a common start state.
2

Background

This section presents the background material necessary to understand the dissertation. In section 2.1, the concept and requirements of real-time systems are defined. Section 2.2 describes the concept of ad-hoc mobile networks and partitioned communication in them. In section 2.3, view-synchronous group communication, a category of multicast protocols, is described. Section 2.4 gives a brief overview of the theoretical aspects of database systems. In section 2.5, an overview of data replication is given. Section 2.6 describes how the PRiDe replication protocol works.

2.1 Real-Time Systems

Data systems can, for example, in emergency response scenarios such as the one described in the introduction, be accessed and updated by humans and systems that require timely and predictable response times. Timeliness and predictability of computer systems is studied in the field of real-time systems.

In real-time systems as opposed to traditional computer systems, the value of a service is not only dependent on the result, but also on the timeliness of its delivery. For example, the correct decision to change a railway switch can have disastrous effects if it is executed when a train is on top of it. Timeliness requirements can, for example, come from a user interaction that should feel responsive or a safety system in a nuclear reactor that must react timely to prevent meltdowns.

A system must be available to be able to produce a result. This fact in combination with the required timely delivery of results makes the system availability a key issue for real-time systems. One way to improve the availability is to add redundancy, for instance having more than one provider of a critical service. A natural way to design redundant systems is to use replicated computing nodes that are connected in a network. Several nodes can provide the same service. If one fails then the other can still provide the service of the lost node. To guarantee that all such replicas provide equal responses two different approaches can be used (Coulouris et al. 2005, p. 618); (i) active replication where all replicas process all requests to end up in the same state or (ii) passive replication where the replicas exchange state information to synchronise their state.
2.2 Mobile Ad-Hoc Wireless Networks

Mobile ad-hoc networks (MANET) are a way to connect mobile computing nodes in a network without any dependency on existing stationary networking equipment. In emergency response scenarios, existing stationary communication equipment may be destroyed or non-existing. In such situations support for ad-hoc network, mobile or not, can provide the only way for data communication until a support infrastructure is set up.

In mobile ad-hoc wireless networks, each node connects to the nodes that are within their communication range. A graph can be used to describe the topology of the network. The communication devices are represented as graph nodes, in this thesis referred to as nodes. Edges between nodes represent that they are within communication range of each other. Some situations, for instance due to power saving, may lead to one way connectivity between nodes. One way communication is not regarded here since power saving strategies are beyond the scope of this thesis. Two nodes can communicate if there is a path between them, in other words when the traffic can be routed between them by the nodes in the path.

A network is partitioned if there are two or more nodes that cannot communicate with each other. A partition consists of the non-empty set of nodes where all nodes can communicate with each other. Two network partitions are long-lived with regard to each other if there is no known upper bound on the time they stay separated. Long-lived partitions may appear in all wireless networks where the nodes are allowed to move in an unbounded way. There is nothing that hinders a node from wandering away indefinitely, due to the boundless movement, forming a separate partition.

In a network with stationary nodes topology changes are often due to nodes that connect or disconnect. In wireless networks with mobile nodes the topology changes dynamically as node movement and other factors can move nodes in and out of reach of each other. Information can travel between two partitions as nodes with the information reach nodes of the other partition. Data can in some cases travel between two nodes even though they stay partitioned from each other at all time. For example, in figure 2.1 the set of nodes is represented as a set of vertices \( V = \{n_1, n_2, n_3, n_4\} \) and the set of edges at time \( t \) and \( t + 1 \) is \( E_t = \{(n_1, n_2), (n_3, n_4)\} \).

The initial network configuration is shown at time \( t \), at time \( t + 1 \) node \( n_1 \) sends a message, illustrated as a circle around the node, to \( n_4 \). This message cannot be routed since there is no path between them. At time \( t + 2 \) the set of edges is \( E_{t+2} = \{(n_3, n_4)\} \) and the message can still not propagate to \( n_4 \). At \( t + 3 \) the set of edges is node \( E_{t+3} = \{(n_2, n_1), (n_3, n_4)\} \) and the message can be propagated to \( n_4 \) from \( n_2 \). The nodes, \( n_1 \) and \( n_4 \), did not belong to the same partition at any time, but could still communicate indirectly since the message is stored by \( n_2 \).

2.3 View-Synchronous Group Communication

View-synchronous group communication (Moser et al. 1994, Coulouris et al. 2005,
Schiper & Sandoz (1993) is a group name for strategies for reliable multicast in unreliable asynchronous networks, that is, networks with no delivery guarantees and arbitrary timing behaviour. It is reliable in the sense that the sender is guaranteed that all reachable nodes receive the message. Such guarantees are impossible in asynchronous networks (Fischer et al. 1985) but can be held if virtual synchrony is introduced in the network. Nodes in the network use an imperfect failure detector to introduce virtual synchrony. The failure detector marks nodes as suspected to be unreachable if they fail to acknowledge messages within a specified time.

The protocol uses views to indicate membership changes. A view is the set of nodes that are members of a communication group that has not been excluded by the failure detector. In this thesis, the views represent the partitions and all nodes that can communicate strives to become members of the same view. These views can be used both to give the individual nodes information of which nodes that are member of the same partition and to provide guarantees on the delivery of network messages. View-synchronous communication guarantees the following properties for message delivery (Coulouris et al. 2005, p.612):

**Agreement** Correct members of a view, $v_1$, install the same next view, $v_2$. If one member of $v_1$ delivers a message before it installs $v_2$ then all members of $v_1 \cap v_2$ deliver that message before installing $v_2$.

**Integrity** A correct node delivers a specific message at most once.

**Validity** The node that issued a message delivers that message. If a view member fails to deliver the message within the same view as the sender then the it will not share the next view with any member of the view that deliver the message. We require, for simplicity of presentation, that the sender delivers the message in the view it was sent.

To make these guarantees the reception of a message by the communication layer is separated from the delivery of the message to the application by the communication layer. Messages are delivered from nodes that are members of the view. A node is member of a view from the point when that view has been installed at the node. No messages addressed to a view are delivered until the node has installed the view. The node is no longer member of a view from the point it has installed another view. Babaoğlu et al. (1998) describe a view-synchronous group communication protocol with support for network partitions in which nodes that stay connected

Figure 2.1: Data propagation in a partitioned network
eventually becomes members of the same view. We relax this requirement to allow
the nodes to voluntarily form smaller, possible singleton, partitions as a means
to reduce resource usage. In singleton partitions the nodes own messages can be
delivered immediately and the network can be disabled.

\[ \text{Figure 2.2: Allowed and disallowed message deliveries (adapted from Coulouris et al. 2005, p.613)} \]

Figure 2.2 shows a variety of possible communication scenarios. Each node,
\( n_i \) – \( n_3 \), has a horizontal arrow that marks their timeline. The angled part of \( n_i \)'s
timeline indicate that it is either, when the distance to the other increases with time,
disconnected from the other or, when the distance decreases, reconnected to the
other. The arrows that start at \( n_i \)'s timeline indicate that a message is sent. The
message arrows either end at a cross, indicating that the message is not delivered,
or at the timeline of a node, indicating that the node delivered the message.

Figure 2.2a - 2.2b shows the standard case, all members of the view deliver the
message sent by \( n_i \), the only difference is the group constellation. The scenario in
figure 2.2c is allowed since \( n_i \) that delivered the message do not share the next view
with \( n_3 \) and \( n_3 \) that did not deliver the message. The scenario in 2.2d is not allowed
since the nodes deliver the message in different views. Scenario 2.2e is disallowed
since nodes that are not members of the sender’s view deliver the message. Scenario 2.2.1 is disallowed since we require that the message is delivered in the view it was sent.

2.4 Database Systems

Databases are used for data storage and different types of databases use different units of storage. Databases that store objects are called object-oriented. Object-oriented databases use an object model to define how the data are represented. Khoshafian & Copeland (1986) introduced an object model based on surrogate identifiers. Surrogate identifiers are system generated globally unique identifiers used as unique keys to stored objects. This type of identifier provides a location independent way to access objects. An object o is a tuple \((i, t, v)\), where \(i\) is the surrogate identifier for the object, \(t\) determines the type of the object, \(v\) is the stored value of the object. The database is a set of objects that can be accessed through an access function. The access function is a bijective function, \(I \rightarrow O\), that is each key is mapped to a unique object.

Transactions are a key feature of databases. Transactions are a way to group data operations into an inseparable unit that either executes entirely or not at all. A transaction can itself be viewed as if it is a data operation and can be part of a transaction. Once a transaction has completed it is said to be terminated. If it is successful it is said to be committed, otherwise it is aborted. A transaction should uphold the atomicity, consistency, isolation and durability properties (Gray 1981) (often referred to as the ACID properties (Haerder & Reuter 1983)), and the meanings of the properties are:

**Atomicity** A transaction is an indivisible unit. It is either executed entirely or not at all.

**Consistency** A transaction preserves consistency. If the data are consistent before the transaction is executed it should still be consistent afterwards.

**Isolation** The state of a partially finished transaction should be invisible to other data operations. This ensures that other concurrent operations’ results are not influenced by transactions that are later aborted.

**Durability** A committed transaction cannot later be invalidated. This ensures that other operations that are based on the result of a committed transaction are not later based on invalidated data. For this property to hold in a nested transaction it must also hold for the parent transaction.

The atomicity property requires that a performed update operation that is part of a transaction can be undone later if the transaction is aborted. This can be implemented by storing the state prior to the updated object in a log. The log entry can be pruned if the transaction commits and used to restore the database if the transaction aborts.
A consistency property can be used to constrain the allowed state of the data in the database. The constraints can be represented as a consistency predicate, evaluating true if the database is in a consistent state. A financial database can, for example, require that no savings account has a negative value. In databases with concurrency, two concurrent transactions are often required to have a result as if they were serialised, that is executed in a serial order. This concept can be generalised to distributed databases. Transactions are one-copy serialised if the result is the same as if they were serialised on a single copy of the database (Coulouris et al. 2005, p. 641).

Real-time databases need to have predictable worst case response times. Databases have traditionally focused on high average throughput; in real-time systems, where one missed deadline can be disastrous, it is more important to improve the worst case. Possible ways to make the response time more predictable is to use a deadline aware scheduler and to use a main memory resident database (Eriksson 1998).

An object-oriented database can be replicated over several computing nodes forming a distributed object-oriented database. A replication protocol is used to keep the replicas synchronised. The replication protocol often tries to hide the fact that the database is replicated and behave as if all access is done on a single virtual database. The database is fully replicated if every node contains a replica of each object stored in the database.

2.5 Replication and Consistency

As mentioned, replication protocols are used to synchronise divergent replicas. Two previous equal database replicas are in different, but not conflicting states, if one of them is updated but not the other. They are in conflict if both of them, independently, have performed updates based on a previously common state. In the first case, the replica without the update can be updated to reflect the update of the other. In the second case, the conflict must be resolved before the replicas can be synchronised to a common state.

The strategy used for data replication in distributed database systems can be classified along two dimensions according to how they handle conflicting updates (Davidson et al. 1985); pessimistic—optimistic and syntactic—semantic. The first dimension defines when conflicts are detected and the second how they are resolved.

Pessimistic approaches only allow transactions to commit if it is possible to guarantee that they do not cause any inconsistency. For example, the two-phase commit protocol (Coulouris et al. 2005, p. 570) first coordinates the nodes and if all nodes can agree on the result they commit in the next phase, otherwise they abort. This guarantees consistency since the nodes only agree if the commit do not result in an inconsistent state and the transaction is only committed if all nodes agree. However, if the system uses asynchronous communication then this distributed consensus falls under the impossibility result of Fischer et al. (1985).

Optimistic approaches allow transactions to proceed before all nodes has guar-
anteed that the transaction does not cause conflicts. Conflicts are resolved later as they are detected. The resolution can either be performed with a backward or a forward strategy. Backward resolution algorithms undo some of the conflicting transactions until the state is conflict free. Forward resolution keeps the conflicting transactions but transforms the database to a consistent state (Saito & Shapiro 2005). Alternatively, the algorithm can resolve conflicts per object instead of complete transactions, for instance data-centric as opposed to transaction-centric resolution (Phatak & Badrinath 1999). Detached replication is a version of the optimistic approach where all operations are done locally and conflict resolution and replication are done asynchronously in the background (Gray et al. 1996). This strategy can improve the predictability of the commit times since it does not depend on any network communication to finish the update.

The syntactic—semantic dimension defines what type of information that is used to define the correctness criteria. Syntactic strategies do not use the semantics of the operations in the transactions when defining the correctness criteria. Semantic strategies define correctness for the data, either as transaction serialisability or by some correctness criteria for the stored data. A semantic strategy can be less restrictive than syntactic even if both use one-copy serialisability as the correctness criteria. For example two transactions can be syntactically in conflict but the result of the conflicting operations is independent of the ordering. Hence they are semantically correct.

Version vectors (Mattern 1989) can be used to syntactically detect conflicting updates. A version vector is set \( V \) of tuples \((N, L)\) where \( N \) is a node and \( L \) is a logical clock, often an integer. Nodes increment their logical clock every time they update the database and associate the state of the version vector with that update. The version vector is piggybacked onto all outgoing messages. When a node receives a message from another node the receiving node updates the version vector so the logical clock of each node is set to the maximum of the one in the received vector and the own vector. The vector associated with an update can be compared with updates on the same data on other nodes to determine which one causally predates the other. A version vector, \( vv_1 \), causally predates another, \( vv_2 \), if for each node the associated clock in \( vv_1 \) is less then the one in \( vv_2 \). If neither \( vv_1 \) predates \( vv_2 \) nor \( vv_2 \) predates \( vv_1 \), then they are logically concurrent. This strategy provides causal ordering. All nodes order the updates from a node in the same order as they were issued on that node. Total ordering of all updates are handled in a causal order and the order is the same on all nodes, can be obtained if the set of conflicting updates is handled in the same order on all nodes.

Replication protocols either rely on state-transfer or operation-transfer (Saito & Shapiro 2005). State transferring protocols send an indicator of the objects state, possibly the complete representation of the object, between nodes to propagate changes. Operation transfer means that the protocol propagates the operation which transforms the object to its target state. The resulting state after an operation transfer depends on the initial state and is only guaranteed to be equal if the initial state is equal. State transfer, on the other hand, replaces the initial state with a new one and does thus not depend on the initial state.
We focus on optimistic replication with application specific forward conflict resolution. Forward recovery is chosen since backward conflict resolution is not suitable for real-time systems in networks with long lived partitions. Multiple partitions that are updated for a long time may result in many conflicting operations. With backward recovery conflicting updates are reverted until all conflicts are resolved. This can result in situation where the data go backwards in time to an older state if the now reverted updates have been available to the application. The application must then reapply updates to ensure that the data reflect the current state. This can result in unpredictable work load and to minimise the set of reverted updates is a NP-complete problem (Davidson et al. 1985).

Application specific conflict resolution allows the user to resolve conflicts in a sensible way. For example, two conflicting observations of the position of a damsel in distress can be transformed into a new position in between the two observed positions with high uncertainty. In this case, an optimistic strategy is preferred since we are interested in availability and predictability in unpredictable networks.

It may be necessary to reissue updates that have been reverted by a backward conflict resolution. If the update is coupled with a real-world action then reverting it may result in an incorrect state (Garcia-Molina 1983). For example, if the update counts the number of drilled holes then the database model is incorrect if the update is reverted since the physical hole cannot be undrilled. If the update is reverted then it must be reissued by the application. But it is not always possible to, within reasonable time, inform the issuing application that an update is reverted when an optimistic approach with backward conflict resolution is used in networks with unbounded partitions. For example, suppose a node issues an update that does not cause a conflict in the current partition. Later on, after the issuing node has left the partition, the partition merge with another partition. The new partition has a conflicting update and one of the updates must thus be reverted due to the backward strategy. If the update from the now unreachable node is reverted then the issuing node cannot possibly be notified. The node can possibly be notified later if it reconnects to a node that contains information about the reverted update, but this may never happen. It is thus impossible to always inform the issuing application if its update is reverted.

There exist several protocols that combine forward conflict resolution with optimistic updates (Andler et al. 2007, Ekenstam et al. 2001, Kermarrec et al. 2001, Terry et al. 1995). Common for all the identified protocols with these properties, with an exception for PRiDe (Andler et al. 2007), is that they are designed for disconnected operation. The nodes update the data while disconnected from the other and later when they are reconnected they synchronise their states. In contrast, PRiDe uses these techniques to improve response times and predictability for real-time tasks in systems where communication failures, such as partitions, are failure mode and not a normal mode of operation (Syberfeldt 2007). All these protocols, except Bengal (Ekenstam et al. 2001), has a storage overhead that is dependent on the number of updates performed between synchronisations. Bengal achieves a constant overhead by only storing the state of each object (Ekenstam et al. 2001) but sacrifices the flexibility the full log of update operations can provide. IceCube (Kermarrec
et al. 2001) and Bayou (Terry et al. 1995) use this information to reorder the updates in a way that minimises the number of conflicts. PRiDe uses information about update operations to, for example, merge concurrent commutative operations. The focus on non disconnected operation for PRiDe limits the value of log reordering to minimise the number of conflicts.

The constant storage overhead achieved by the state transfer approach of Bengal provides a way to limit the resource usage in networks without assumptions on update rate and the time the network stays partitioned. Forward conflict resolution on state alone between nodes in the same partitions can both add communication overhead and reduce the information available for the resolution routines. Within a partition, nodes can communicate in the way PRiDe is designed for but PRiDe must be extended to handle long lasting partitions.

### 2.6 PRiDe

As mentioned, PRiDe (Protocol for Replication in DeeDS) (Andler et al. 2007) is an optimistic and detached replication protocol with eventual global consistency and forward conflict resolution. Figure 2.3 shows the architecture of a system with PRiDe replication. PRiDe makes both the optimistic and the tentative, not yet guaranteed to be stable, state of the replica available to the application programs. After a local update to an object, the tentative result of the update is immediately available via optimistic reads. This provides the benefit of the latest available value but without stability guarantees. For example, an optimistically read value may later be changed when conflicting updates from other replicas reach the local replica. Such conflicts must be resolved equally on all nodes and may change the tentative value before it is made stable. How conflicts are resolved is determined by data specific conflict routines provided by the application programmer. Such routines must be added to each type of object stored in the database. The function takes a set of conflicting updates to an object, resolves the conflict and returns a single update that represents all the conflicting updates.

PRiDe synchronises replica updates into rounds. Each node may issue at most one update per round. A node starts a new round when an update is issued on it and the update operation and the number of the round is broadcast to the view. The receiving nodes increase the number of their next round, skip rounds, if it is necessary to reach the number of the round in the received messages. A special null update is broadcast together with the number of each skipped round.

Received updates are ordered into generations, one generation per round. A generation can store one update per node. The generations are stored in a log ordered by their round. Figure 2.4 shows a possible log from a network of three nodes updating an arithmetic value. The rows illustrate generations and the columns the origin of the stored update. The null update is represented as a diagonal line. This log provides, if no covert channels exist, a partial causal ordering of the update. Since each update issued from a node is given a higher round then that of messages received by the node prior to the update. A generation is regarded as stable once it
Figure 2.3: Architecture of PRiDe

contains an update from each node (generation $g_1$ and $g_2$ in figure 2.4 are stable). Each generation will eventually be stable at each node since each node reliably broadcasts a message with the number of the round once the node know about it. The definition of reliable broadcast ensures that the first message eventually reaches every node and that the resulting responses from each node eventually reach all nodes, stabilising the generation at every node.

Updates in stable generations can be integrated into the database. The stabilisation function in PRiDe iterates through the log and removes stable generations and applies the updates stored in them to the database. But a generation can contain more than one update since multiple nodes can issue updates concurrently. See generation $g_2$ in figure 2.4 for an example of a generation with conflicting updates. Such updates are regarded as being in conflict and must be resolved. PRiDe relies on conflict resolution routines provided by the application. Such routines take a set of conflicting updates and transform them into a single update.

<table>
<thead>
<tr>
<th>$n_1$</th>
<th>$n_2$</th>
<th>$n_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1$</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td>$g_2$</td>
<td>-10</td>
<td>$\times 2$</td>
</tr>
<tr>
<td>$g_3$</td>
<td>$\div 3$</td>
<td></td>
</tr>
<tr>
<td>$g_4$</td>
<td></td>
<td>+1</td>
</tr>
</tbody>
</table>

Figure 2.4: Update generations in PRiDe

PRiDe has been designed with real-time databases in mind. The application can
initiate stabilisation by invoking a stabilisation function when it does not interfere with its real-time tasks.

PRiDe supports transactions by propagating all updates in the transaction as a single message. Each update is paired with the generation number they belong to. If a transaction is aborted then the local changes are undone and the other replicas are informed that no updates were performed on the replica. Once a transaction has begun, all updates from other nodes are queued and the state of the database is only changed by the operations that are part of the active transaction. Since no update messages are sent to the other nodes they cannot stabilise generations created after the transaction begun until it is ended.

The combination of data-centric forward conflict resolution and optimistic transactions can result in ACID anomalies. For example two transactions, $x_1$ and $x_2$, are concurrently executed on different nodes. Both transaction update object $\alpha$ and $\beta$ such that the result after $x_1$ is $\alpha_1 = 5$, $\beta_1 = 1$, and after $x_2$, $\alpha_2 = 1$ and $\beta_2 = 5$. The application provided conflict resolution policy for both objects is to keep the update with the lowest value of concurrent updates. The final result, $\alpha = 1$, $\beta = 1$, is constructed from half of $x_1$ and half of $x_2$ clearly not compliant with the atomicity property. The application can provide assertions with the transaction to handle this type of situations. The assertions can evaluate the stabilised result and trigger compensating actions if it is not the expected. A possible approach is to create a compensating action that reverts the transaction and retries to commit it later. However, a later transaction can read the tentative value of the previous transaction before it is reverted and this violates the isolation property. The consistency property depends on how the provided conflict resolution handles conflicts; a function can resolve a conflict into an update that results in an inconsistent state.
Partitions in Distributed Real-Time Databases

This chapter gives the aim of this thesis and motivation for why it is of interest to investigate this aim. Following the motivation is a definition of the thesis problem and necessary properties of a problem solution, followed by a list of assumptions made when designing the solution.

3.1 Aim

The aim of this project is to investigate how arbitrarily long lived partitions can be managed in distributed real-time databases. The primary focus is management of long-lasting partitions in mobile ad-hoc networks with application specific conflict resolution. To limit the scope of the project no considerations is taken to power management issues such as one-way communication modes and traffic balancing between nodes.

3.2 Motivation

Replication protocols must handle long-lived network partitions when they are used in mobile wireless networks with unbounded node movement (see section 2.2). Such networks are used for real-time collaboration in disaster and battlefield situations where it is impossible to rely on backbone infrastructure (Hong et al. 1999). A shared whiteboard architecture built on a distributed real-time database can facilitate such communication (Syberfeldt 2007).

PRiDe is developed for replication in such real-time databases. This makes PRiDe suitable for the communication within existing partitions (see section 2.5). However, an extension that adds support for long-lasting network partitions must be designed before PRiDe can be used in mobile wireless networks.
3.3 Objectives

To accomplish the aim the task is broken down into the following objectives:

- **To implement PRiDe on top of an existing embedded database**
  
  A reference implementation of PRiDe is needed both as a base for the extension and as a reference during the evaluation.

- **To design and develop an extension to PRiDe that adds support for long-lasting network partitions**
  
  PRiDe should be extended to use a group communication strategy to handle membership changes in partitions. The synchronisation strategy must be extended to be able to resolve conflicts in non tentative updates. Such extension must, in order to not inhibit the progress and introduce sources of unpredictability, be designed to handle the following subproblems:
  
  - **Continued service when the network topology changes**
    
    The level of service should be constant; the application programmer should potentially be able to safely ignore changes in the network topology. The application programmer must provide a function that merges a set of conflicting versions of each stored object type.
  
  - **Eventual consistency between nodes that stay connected**
    
    All replicas that stay connected should eventually reach a common state if they cease to update the database.
  
  - **Predictable resource usage**
    
    The partitions management must have predictable processor, network and memory usage. The upper bound on resource usage must not be dependent on how long the network stays partitioned.

- **To compare and contrast the extended PRiDe with existing partition aware replication protocols**
  
    The extended PRiDe should be compared with other partition aware replication protocols. Advantages and disadvantages should be identified to place this work in a larger perspective.

3.4 Assumptions

To focus on the replication problem rather than the partition detection and data routing issues the following assumptions are made.

- The database is fully replicated; every node has a replica of each object in the database.
• The set of stored objects is static, it is possible to update but not to add new or remove existing objects.

• The protocol can leverage on an existing view-synchronous group communication subsystem.
An extension to PRiDe that adds support for long-lasting partitions is proposed and evaluated. The evaluation is both based on theoretical arguments and tests of a proof-of-concept implementation. The extended PRiDe is compared to the unextended version to verify that the extension does not change the behaviour of the protocol unless the network is partitioned. PRiDe is implemented in the same framework as the extended version to facilitate this comparison.

The behaviour comparison between PRiDe and the extension is performed under a simulated database load. A simulator is developed for this evaluation. The simulator can both simulate networks with mobile nodes and static networks. Static networks are necessary for comparisons with PRiDe and mobile nodes are necessary to validate that the extension works as expected.

The simulator is not developed to simulate a real scenario. This is because the protocol is not targeting a specific scenario. Without a target scenario no realistic data from mobile wireless ad-hoc networks can be collected. The simulator does therefore not have the data necessary to simulate realistic communication delay, communication error and the node movement. Node movement is instead arbitrarily chosen to provide a network with frequently occurring network partitions and all communication happens instantaneously. Timing measurements from the simulation do thus not show any realistic values and are of little use.

### 4.1 Implementation of a database with PRiDe replication

Prior to this work there existed no implementation of PRiDe on top of a database (Holmgren 2006). Section 4.1.1 describes two approaches to how PRiDe can be implemented on top of an existing database. The different parts of the chosen architecture are depicted in figure 4.1. Detailed description of the different part is given in the following subsections, the overall architecture in section 4.1.1, the object store in section 4.1.2 and pPRiDe in section 4.2.
4.1.1 Architecture

The replication protocol is designed as a layer between the database and the application. This is one of two proposals in Holmgren (2006) for how replication can be implemented atop of an existing database. The other is not used because it requires tighter coupling with the database. The chosen approach must map the object oriented PRiDe interface to the interface of the underlying database. This mapping adds complexity to the implementation that can be reduced if a database with an interface similar to that of PRiDe is used. For this reason we developed the system on top of Berkeley DB\footnote{http://www.oracle.com/technology/products/berkeley-db/}. Berkeley DB is an open source key-value database engine with transaction support. Both the key and the stored value can be an arbitrarily chosen data in serialised form. In this chapter serialise, serialised and deserialisation means to create a form that can be stored in a single continuous chunk, to be in such form and to recreate the original object from such form. The concept should not be confused with serialisability of database transactions.

An object interface is added on top of the key-value database. This object interface is based on the surrogate object model (see section 2.4). Integers are used for the surrogate identifiers. For the object type a class identifier is used. The object state is used for the value field. There is no mechanism for generating unique ids in the prototype. This is basically because it is assumed that the set of objects are predetermined and cannot be changed. But, for example, UUIDs (ISO 2005) can be used to generate unique series from a unique identifier. The type-field is represented as a class identifier, an integer, which connects the stored data with its class. The class provides the interface of the object. The state of the object is stored as the value
in the object model.

### 4.1.2 Stored Objects

The database is designed to store C++ objects. The classes of the stored objects must have a function that serialises the object, a function that provides the size of the serialised form and be given a unique identification number. The database stores the objects in their serialised form. An object factory is used to restore the objects upon retrieval. The factory has an object deserialisation routine register for each class identifier. The deserialisation routines are used to restore stored objects into an object instance.

Operation transfer is implemented by propagating update programs between the replicas. An *update program* is a way to represent updates. Each method that may change the state of the object must return an update program to the replication protocol. The update programs are objects that store the data necessary to replay the update on another replica. Each class stored in the database must implement a function, *update*, that takes update programs as its only argument. Calling this function with an update program should have the same effect as performing the operation represented by the update program.

For example, a class that stores a colour can have an *updateSaturation* function. We have two replicas of the object, α and β, both with the same state. If we use the *updateSaturation* function of α to update its saturation, then the states of α and β differ. The *updateSaturation* function returns an update program, u. If we use the *update* of β with u as argument then both α and β will again have the same state reflecting the updated saturation.

### 4.2 pPRiDe

*pPRiDe*, partition aware PRiDe, is the extended version of PRiDe developed in this thesis. pPRiDe is a hybrid protocol that uses both state transfer and operation transfer. The extension adds state transfer as a mean to integrate nodes into an existing partition without the need for a common initial state. To handle conflicting states, the user must, in addition to the routines for resolution of update conflicts required by PRiDe, also provide routines for conflicting state transfers from node integration.

PRiDe requires messages addressed to a generation from all nodes in the network before the generation can be stabilised. This requirement must be relaxed for the protocol to progress in networks with long-lasting partitions. Without a change some generations may never stabilise since nodes may be partitioned forever and the necessary messages may thus never reach all nodes. To store all non stabilised generations would violate the goal of predictable resource usage. Generations cannot be stabilised when a single node has left a larger partition and formed a single node partition. This single node cannot be assumed to ever reconnect to the other nodes due to the properties of the assumed network. Nodes may update the database when
the single node partition exists, due to the design goal of the protocol. PRiDe creates a new generation every time a node makes an update. Thus the resource usage cannot be bounded unless the requirement for stabilising generations is relaxed.

A generation in pPRiDe can be stabilised when it contains a message from every node that has continuously been reachable from the stabilising node since the creation of the generation and until it can be stabilised. This guarantees that the result of a stable read is conflict free amongst the members of the current perceived partition. The perceived partition is provided to pPRiDe by a partition aware view-synchronous group communication protocol (see section 2.3). The communication protocol is responsible for group communication and for managing membership of dynamically changing partitions. Nodes that can communicate should eventually become members of the same view. How the view-synchronous group communication layer is integrated is shown in figure 4.2 which can be compared to the standard pride depicted in figure 2.3.

![Architecture over pPRiDe](image)

Figure 4.2: Architecture over pPRiDe

### 4.2.1 Network Partitions

The view-synchronous group communication protocol provides a mean for the node to get information about what nodes are members of its partition. The communication protocol installs views on each node to reflect membership changes. Each view contains a unique identifier and a list of nodes that are part of the same view.
Members of the same view can use a communication protocol to communicate with each other and the protocol makes delivery guarantees (see section 2.3). The set of members in the installed views are used by pPRiDe to reduce the list of nodes that must send update messages before a generation is regarded as stable.

pPRiDe adds hooks and methods for handling view changes to PRiDe. A view can accept new members that previously were members of other views and members may also leave the view. A state transfer mechanism synchronises the state of new and old members to a common state when new members have been accepted. State transfer is necessary to guarantee that nodes can be integrated into the partition. The reason for this is that nodes that join the partition may have an arbitrary state. Pure update transfer requires a common baseline state to guarantee that the transferred updates result in a common state. Such baseline state is obviously not available without a state transfer if updates has been merged and pruned.

Nodes that join a partition may have different conflict set and generation numbering than the existing members. To integrate these nodes all nodes install a common view and derives a common initial state to perform updates to and a common generation numbering. This is solved by using a conflict set per installed view. Two special generations are added to the conflict set, the initial generation and the final generation. The initial generation is used to derive the initial state from the final states of all members’ previous view. It predates all other generations associated with the view. The final generation is used to detect when all nodes of a view have reached a common final state. It is predated by all other generations of the view. A node sends a message informing the others that it thinks an old view can be finalised when all its generations except the final is stable. This notification is added to the final generation. Each node sends a state message addressed to the initial generation of the next view when the final generation is stable. The initial state for next view is derived from all the state messages when the initial generation is stable. The application programmer provides specific state conflict routines that are used to derive this initial state.

This state transfer can result in situations where an updated version is merged with a prior version. For example, when two nodes are part of the same partition and thus share the same state for an object. Then the nodes get separated into two different partitions and one updates the object and the other one does not. Later, when they reconnect into a new partition consisting of the two nodes, they merge the state of the object. One possible strategy is to use the average value of the two states, but this dilutes the result of the update. This problem can be reduced if all objects store a version vector (Ekenstam et al. 2001). When a node updates an object it increments its logical clock in the version vector that is stored in the updated object. If the version vector of one of the states predates that of another then its state can be ignored in favour of the other.

### 4.2.2 Update propagation in pPRiDe compared to PRiDe

In this section we show the sequence of method invocation started as a response to an update. The sequence starts with a single node that issues and propagates an
update and it continues at all nodes that receive the update. The view-synchronous communication used in pPRiDe requires some changes to this update propagation. The strategies for both PRiDe and pPRiDe are presented and differences are highlighted.

The view-synchronous group communication used for pPRiDe has separated reception and delivery to the propagated update messages. Figure 4.3 and 4.4 compares how update messages are handled in pPRiDe to how they are handled in PRiDe.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>An update is issued</td>
</tr>
<tr>
<td></td>
<td>(a) append the update to the log</td>
</tr>
<tr>
<td></td>
<td>(b) \texttt{vsend} the update</td>
</tr>
<tr>
<td>2.</td>
<td>The update message is received but only processed if the sender may address the target view</td>
</tr>
<tr>
<td></td>
<td>(a) Create missing generations that precedes that of the update</td>
</tr>
<tr>
<td></td>
<td>(b) \texttt{vsend} a no update message for each new generation</td>
</tr>
<tr>
<td></td>
<td>(c) Inserted the update into the target generation</td>
</tr>
<tr>
<td>3.</td>
<td>The update is delivered</td>
</tr>
<tr>
<td></td>
<td>(a) If the sender may address the target view</td>
</tr>
<tr>
<td></td>
<td>i. Execute step 2 if the update is not in the log</td>
</tr>
<tr>
<td></td>
<td>ii. mark the update as delivered</td>
</tr>
</tbody>
</table>

Figure 4.3: Update in pPRiDe with differences from PRiDe underlined

PRiDe uses two functions, one for to issue an update (step 1) and one for receiving a propagated update (step 2). A third step is added to pPRiDe to handle the message delivery from the view-synchronous group communication layer. The steps are executed in the given order for a specific update, one node issues an update at step one. This action triggers action number two on all reachable nodes and for pPRiDe later also step three.

Step one, the update function, only differs on one point. The two versions use different primitives to send the update message to the other nodes. pPRiDe uses \texttt{vsend}, the send primitive provided by the view-synchronous group communication protocol, and PRiDe uses a reliable broadcast.

Step two is executed when the update sent in step one is received at a node. The steps $2a-2c$ in PRiDe maps to steps $2a-2c$ in pPRiDe again with the defined
view and send primitive as the only change. Step 3 in pPRiDe is added to not allow communication to historical views between nodes that do not share history. Messages addresses to future, not yet installed, views are also filtered out.

Step three is executed when the delivery properties of the view-synchronous group communication is guaranteed. This step is obviously unique for pPRiDe. It marks a previously received update as upholding the delivery guarantees, step 3(a) ensures that the update has been inserted into the log.

The update propagation has, as shown, only been changed slightly in the transition from reliable broadcast in PRiDe to view-synchronous group communication in pPRiDe. In addition to update propagation, pPRiDe also uses network communication for state transfers and to agree on a final state between view members. Since these are unique for pPRiDe their use cannot be compared with PRiDe. Their purpose and use are described in chapter 5.

### 4.3 Simulator

The network simulator used for validation is designed to simulate three different aspects of the environment for a distributed database in a mobile ad-hoc network; movement, communication and database load. To make it possible to evaluate if the nodes converge to a common state the simulator is designed to give visual feedback on the state of the different nodes. Figure 4.5 shows a visualisation of the simulator. The colour of the node represents the state of its replica of the colour object. Overlapping nodes, visualised with equally coloured outer rings, can communicate.
4.3.1 Node Movement

The nodes are placed in a two dimensional arena. Each node is modelled as a circle with a radius that represents the signal strength of its radio. Two nodes can communicate directly with each other if their ranges overlap. Two nodes can communicate if they either can communicate directly or they can communicate with a node that can communicate with the target node. The nodes move with a constant speed but change their direction. The direction is changed by updating the third derivative of the heading from a random variable with normal distribution. To hinder the nodes from wandering so far away from each other that no communication is possible a weak attractor is placed in the space. The attractor is implemented as a force that pulls the node towards the centre of the arena. The strength is proportional to the distance from the centre. This movement pattern does not try to mimic any real pattern. It should be changed to one that, for example, mimic the movement of

4.3.2 Communication

The communication path between two nodes is either perfect or missing. If there is a path of nodes with overlapping communication range between two communicating nodes, then all sent messages arrive unchanged in unchanged order to the destination. The communication is either instantaneous or delayed a specified random amount of time; the same is true for the delivery of messages.

The view-synchronous delivery has in addition to the, agreement, integrity and validity, properties described in section 2.3, has also the necessary uniqueness and exclusion which are further described in section 6.1.
4.3.3 Database Load

Database load is simulated by letting each node randomly perform different updates to different objects with random data. The simulator provides functions to trigger update and stabilisation of the database randomly with normal distribution. Normal distribution is used because we assume that every update is triggered by the human user and is the result of several independent random aspects. The user defines the mean and variance and functions that generate an argument to the update functions to be triggered. No support for simulating transactions is implemented in the simulator. For example, the database store a colour object and we wish to update the hue. To do this we connect the updateHue function to a normal distribution with mean 3 second and variance 0.1 second and provide a function that generates an angle to add to the colour’s hue. When we start the simulator we provide a mean and variance to the normal distribution for the stabilisation.
This chapter gives a description of how the prototype of pPRiDe is implemented. Transaction support has been left out for the sake of clarity. Transactions can however be implemented in the same manner as described in Syberfeldt (2007).

The prototype uses a three layer architecture depicted in figure 4.2. The application can read and update objects and initiate stabilisation of stable update generations. The pPRiDe layer sends messages to the current view of the node through the view-synchronous group communication layer via the non blocking vsend function. The communication layer does in turn deliver messages in accordance to the guarantees described in section 2.3 via the deliver function and possibly earlier without guarantees via receive. The view function is used when new views are installed.

5.1 Communication messages

The implementation of pPRiDe uses three types of messages, Prop, Final and Init, to communicate between the nodes. Final and Init are additions made for pPRiDe. Each of the messages can both be received by a node and delivered by the view-synchronous communication layer. Each message contains information about who sent it, the id of the target view and the key of the object in question. In PriDe only the key is used. The different messages have the following purpose and data:

**Prop** is used to propagate updates amongst the nodes. These contain, in addition to the common data, a generation number and the data describing the update operation. The prop message has the same purpose as the propagation message in PriDe.

**Final** is used to inform the other members of a view that the issuing node will not perform any more updates to that object in that view.

**Init** is used to send the previous views’ final state to a new view so it can be initialised.
In the pseudo code we create messages with a list of attributes as argument. For example `Prop(key, node, view, generation number, update)` the first three arguments are common for all message types. `Prop` messages are created as in the example, `Final` do not take any extra arguments and `Init` messages takes a state representation as the fourth argument. We use dot notation to access the given attributes. For example, the target view is accessed as `msg.view`. The key, sender and, for the `Prop` message, the generation number are accessed in similar manner. We use $M$ to denote the set of messages. We will continue to use capital letters to denote sets and the same letter in lower case to denote members of the set. A message will thus be denoted $m$.

## 5.2 Data Structures and Basic Functions

In this section we describe the data structures and functions used in the later pseudo code presentation of pPReDiE. The generations and generation series inherited from PRiDe (see section 2.6) are augmented with the initial and final generations (see section 4.2.1). A structure to support the notion of installed views and view members are introduced.

We use $N$ to denote the set of nodes in the network and $n$ denotes a node in $N$. The node of the process that executes protocol is marked as $\hat{n}$. The database is denoted with $D$. The set of possible object states for objects in the database is denoted $O$ and $K$ is the set of keys to objects in the database. Objects are retrieved from the database with function $\text{get} : D \times K \to O$ and inserted or replaced with $\text{put} : D \times O \times K \to D$. Each object in the database supports a set of update operations. We denote the union of all such sets $U$. $U$ also includes a null operation $\bot$ that is an identity function on $O$. An object is updated with $\text{performUpdate} : O \times U \to O$.

Updates to the database are stored in generations, a generation is a set, $g = \{(n, u)\}$, where $n \in N$ and $u \in U$. The set is constrained to contain at most one pair for each $n$. The set of generations is denoted with $G$. We use a bracket notation, $g[N] \to U$, to address the $u$ associated to each $n$. We use $g[N] \leftarrow U$ to associate a value to $n$. The current association, if one exists, is replaced with the new. Each generation is in one of three states: unstable, stable or stabilised. A generation is initially unstable. The state of a generation is changed to stabilised with $\text{markAsStabilised} : G \to G$ when it have been integrated into the database. This transition require that the generation first is stable, the condition for a transition from unstable to stable is given later. The $\text{stabilised} : G \to \mathbb{B}$ function tests if a generation is in the stabilised state.

The generations are stored in generation sequences, a generation sequence, $gs = \{g_n\}$, is a sequence of generations where $g_n \in G$. We use $GS$ to denote the set of generation sequences. We use bracket notation to access elements by their position, $p \in \mathbb{N}$, in the sequence, $gs[p] \to G$. The order of two generations that share generation sequence can be evaluated with $g_x < g_y$. This statement evaluates to true when $g_x$ precedes $g_y$ in the generation sequence. New generations are added to the end of the sequence with $\text{append} : G \times GS \to G$ function. The last added generation, $g$, can be accessed with the $\text{last} : GS \to G$ function. A generation series is regarded
as active when they are created. Once the algorithm determines that the generation series is not to be used any more it deactivates it with deactivate : GS → GS. The oldest active generation series for a specific key is accesses with active : K → GS.

The data structures and function described so far are shared by PriDe and pPRiDe. The notion of views used in pPRiDe requires some additional data structures and functions. Let V be the set of views, each node has exactly one view installed at any time. The view currently installed by the node of the executing process is denoted $\hat{v}$. The current and prior views installed by the executing node are ordered into a view sequence, $\bar{v}s = \{v_n\}$ where $v_n \in V$. A function, $m : V \rightarrow P(N)$, map the view to the set of nodes that are members of that view. We use $v_x \not\subseteq v_y$ to evaluate the relative order of two views. It evaluate to true if $v_x$ has the same or an earlier position in $\bar{v}s$ as $v_y$ and to false otherwise. Each view is associated with a generation series per key through a conflict set, $cs : V \times K \rightarrow GS$. We say that a view, $v_1$, supersedes another view, $v_2$, if $v_1$ is positioned after $v_2$ in $\bar{v}s$. The superseded : $GS \rightarrow \mathbb{B}$ predicate test if the view with which $gs$ is associated been superseded by a later installed view. The excluded : $M \rightarrow \mathbb{B}$ function takes a message and tests if the sender of the message is member of all views in $\bar{v}s$ that supersedes the view addressed by the message. The function evaluates to false if the target view is not in $\bar{v}s$.

The protocol uses vsend to reliably send a message to the current view through the view-synchronous group communication layer. This send function is non blocking and control is returned to the caller without any network related delays. The communication protocol provides two reception functions, one without delivery guarantees called receive and one with delivery guarantees called deliver. We mark a generation, $g$, that contains a delivered update from a node, $n$, with deliver : $N \times G \rightarrow N$. We use delivered : $N \times G \rightarrow \mathbb{B}$ to check if an update from $n$ has been delivered to $g$.

The initial and final generation introduced in section 4.2.1 are denoted $g_i \in G_i$ and $g_f \in G_f$. The initial generation, $g_i = \{(n,o)\}$, has the same properties as the other generations except that instead of update it associate an object state, $o \in O$, with the nodes. The final generation, $g_f = \{(n,b)\}$, where $b \in \mathbb{B}$ associates nodes to a boolean value. These generations are not inserted into the sequences in the generation series but are instead related to them through two functions, initial : $GS \rightarrow G_i$ to access the initial generation of a generation series and final : $GS \rightarrow G_f$ to access the final generation. Let $G^+$ be the set of all instantiated generations including all instances of $g_i$ and $g_f$. Each $g \in G^+$ are associated to one and only one generation series. Each generation series, $gs$, is associate to one and only one $v \in V$. This limitation allow us to define a function, $vog : G^+ \rightarrow V$, that maps each generation to its view.

We use this function to define when a generation change state from unstable to stable. A unstable generation, $g$, change to the stable state when delivered : $N \times G^+ \rightarrow \mathbb{B}$ is true for each node, $n$, in the intersection of all node sets, $m(v_n) \cap m(v_{n+1}) \cap \cdots \cap m(v_{n+m})$, where $v_n = vog(g)$, $v_{n+1}$ is the successor of $v_n$ in $vs$ and $v_{n+m}$ is the last of $vs$. The stable : $G^+ \rightarrow \mathbb{B}$ function test if a generation is stable.

The resolveUpdateConflict : $U^{[N]} \rightarrow U$ function takes set of pair of conflicting
updates and the node the update originates from as argument and uses the user-provided resolution routines to resolves them into a single update that is returned. State conflicts are resolved by \( \text{resolveStateConflict} : G_i \rightarrow O \) takes an initial generation and derives an object state from each delivered object state in the generation.

### 5.3 pPRiDe

In this section the pseudo code for pPRiDe is given. The code is separated into eight separate functions that are either used as a response to messages from other nodes or directly by the application program. The application program can use the \( \text{optRead}(\text{key}) \) to read optimistic state of an object replica, update an object with \( \text{update}(\text{key}, \text{updateProgram}) \) and integrate stable generation with the \( \text{stabilise} \) function. The other functions should be called as response to network messages as listed in table 5.1. The \( \text{rcvProp}(\text{Prop}) \) function should be called when a \( \text{Prop} \) message is received, later when the same message is delivered the \( \text{dlvProp}(\text{Prop}) \) function should be called. The other two message types, Final and Init, only have functions to be called when they are delivered, \( \text{finalise} \) for Final and initialise for Init.

<table>
<thead>
<tr>
<th>receive</th>
<th>deliver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop</td>
<td>( \text{rcvProp} )</td>
</tr>
<tr>
<td>Final</td>
<td></td>
</tr>
<tr>
<td>Init</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Function invocation for each message type and the receive and delivery event from the view-synchronous group communication layer

input : \( k \in K \), key to a database object

\( u \in U \), update applicable to the object

instantiate a new generation, \( g \)

\( g[\hat{n}] \leftarrow u \)

\( gs \leftarrow \text{cs}(\hat{v}, k) \)

append(\( g, gs \))

\( \text{vsend}(\text{Prop}(k, \hat{n}, \hat{v}, \text{ generation number of } g, u)) \)

Function \( \text{update} \)

The \( \text{update}(\text{key}, \text{update}) \) function is used to update on an object in the database. The changes to this function in comparison to PRIDe is that it selects the current conflict set from the view set and the use of vsend. It creates a new generation with a generation number that is one higher than the previous highest numbered known generation in the view. This information is then sent, with the vsend function, to all nodes in the view. The vsend function is non-blocking.
meaning that the control is immediately returned to the caller and is not blocked while waiting for the message to propagate over the network.

```plaintext
input : m, a received Prop message
if ¬ excluded(m) then
  gs ← cs(m.view, m.key)
  while m.generation > |gs| do
    instantiate a new generation, g
    g[hat] ← ⊥
    append(g, gs)
    vsend(Prop(m.key, hat, m.view, generation number of g, ⊥))
  end
  g ← gs[m.generation]
  g[m.sender] ← m.update
end

Function rcvProp
```

The rcvProp(Prop) function handles the reception of propagation messages. First, the function checks if the message is addressed to a known view and that the sender has not been excluded from view later than the one addressed. If this is true, then all generations that precede the received update in the targeted gs are instantiated and inserted. For each instantiated generation a new propagation message with the ⊥ update is sent to the view to inform the other members that the node has not issued any update in the instansiated generation. When this is done the targeted generation is guaranteed to exist and the received update is inserted into it. The corresponding function in PRiDe only use the key to select the correct gs and do not need to perform the check in the if statement.

```plaintext
input : m, a delivered Prop message
if ¬ excluded(m) then
  gs ← cs(m.view, m.key)
  g ← gs[m.generation]
  if g[m.sender] = ∅ then
    rcvProp(m)
  end
  deliver(m.sender, g)
end

Function dlvProp
```

The dlvProp(Prop) function is called when a propagation message is delivered from the view-synchronous communication layer. This function first test if the delivered message has been added to a conflict set if not adds it. After that it marks the copy of the message stored in the conflict set as delivered. No function that corresponds to this one exists for PRiDe.
input : none

done ← false

while ¬done and ∃k ∈ K (active(k) ≠ ŷ) do

done ← true

foreach k ∈ K do

gs ← cs(active(k), k)

gi ← initial(gs)

Phase 1

if stable(gi) and ¬stabilised(gi) then

data ← resolveStateConflict(initial(gs))

put(D, data, k)

markAsStabilised(gi)

done ← false

end

Phase 2

if stabilised(gi) then

data = get(D, k)

foreach g ∈ gs ∧ ¬stabilised(g) do

if ¬stable(g) then return

Q = {(n, u)|n ∈ N ∧ u = g[n] ∧ delivered(n, g)}

update ← resolveUpdateConflict(Q)

data ← performUpdate(data, update)

put(D, data, k)

markAsStabilised(g)

end

end

Phase 3

if stabilised(last(gs)) then

if superseded(gs) then

vsend(Final(k, ŷ, active(k)))

end

if stable(final(gs)) then

disable(gs)

vsend(Init(k, ŷ, active(k), get(D, k)))

end

end

end

end

Function stabilise
The stabilise function identifies conflicts and integrates the resolved states into the underlying database. This function should be run by the application regularly to ensure that the log do not grow larger than allowed by the limitation resources. The function performs three phases per object key, marked in the pseudo code. Phase one, integrates the generation series of the oldest active view for the key if its initial generation is stable and it has not been integrated earlier. That is the current object state with the current key is replaced with a new state that is derived from all state representations in the initial generation. Phase two, if the oldest active generation series for the key has been integrated then all stable generation in it are integrated. A generation is integrated by first resolving possible conflicts between all delivered updates in it and then applying the resolved update to the corresponding object in the database. Phase three has two stages, firstly, it test if there exists any updates in the oldest active generation series for the key that has not been integrated. If no such updates exist then it sends a Final message to the view that this is the case. Secondly, it test if the final generation is stable, a positive answer indicates that no more updates will be added to the generation series and we can deactivate it. We can now send a Init message with the current object state to the superseding generation series.

The execution time of the stabilisation function is dependent on the number of stable generations in active generation series with stable initial generation. The update frequency of other nodes does affect the worst case scenario. This either violates goal of predictable execution time or the goal of continued service for all nodes in the partition. To resolve these conflicting goals we allow a node to disconnect itself from the other effectively creating a voluntary partition of one node. In this setting all updates are controlled by the node and the worst case can thus, assuming a predictable application, be predicted. However, no mechanism for this is currently implemented.

```
input : m, a delivered Finalise message
if ¬ excluded(m) then
    gs ← cs(m.view, m.key)
    g ← final(gs)
    g[m.sender] ← ⊥
end

Function finalise
```

The finalise and initialise functions handle the processing of delivered Final and Init messages respectively. The finalise function associates a ⊥ with the sender of the delivered Final message in the final generation of the targeted generation series. This marks the targeted generation series as not being updated by the sender of the message anymore. The initialise function associates the state information from the message with the sender in the initial generation of the targeted generation series. No corresponding functions are used in PRiDe.

When a new view is delivered the view(id, member set) function is used to update the data structures for the protocol. It appends the id to the view set and
\[ m, \text{adeliveredInitialise message} \]

- **Function initialise**

\[
\text{if } \neg \text{excluded}(m) \text{ then} \\
gs \leftarrow \text{cs}(m\text{.view, m.key}) \\
g \leftarrow \text{initial}(gs) \\
g[m\text{.sender}] \leftarrow m\text{.data} \\
\text{end}
\]

- **Function view**

- **Function optRead**

The optRead() function traverses through all generations of all generation series of the requested object. All conflicts are resolved and the updates are applied to a copy of the stable state of the requested object. State conflicts are resolved and replace the copy as they are encountered. The object copy is returned to the caller. The optimistic read function in PRiDe applies updates in the same manner but without the procedure for encountered states.
Evaluation

In this chapter the implementation, pPRiDe, is evaluated against the baseline, PRiDe. We validate that they have equal behaviour in static networks. The behaviour under non static networks with frequent partitions is evaluated to validate that the protocol reach consensus and that the resource usage does not grow over time. pPRiDe is then compared to existing protocols in section 6.3.

6.1 Proof sketch for pPRiDe

This argument is divided into three parts. First, correctness in a static network is shown. Second, correctness when nodes join a partition is shown and third, correctness when nodes leave a partition is shown. Views and partitions are used interchangeably in this section. This is because nodes use views as the only way to know about its partition and what nodes it shares the partition with. A partition-aware membership service in the communication layer ensures that nodes that stay connected in a partition eventually share view. The argument uses the notion of stable generations. The replicated database is assumed to contain a single object and all generations and updates is thus addressed to that object. The argument holds for larger databases too since the protocol handles all objects in isolation. A generation, $g$, in a view, $v$, is stable for a node, $n$, when all earlier generations are stable and it contains a delivered message from each node that shared $v$ and all consecutive views with $n$. That is, $n$ cannot stabilise $g$ when $g$ is missing a message from a member of $v$ that has not been excluded from a later view installed by $n$. The protocol depends on a view-synchronous group communication protocol that provides the following properties:

**Uniqueness** Each view can be uniquely identified. When the node set of a view is changed it is given a new identity that discriminates the new view from all future and historical views.

34
Exclusion If two nodes, $n_1$ and $n_2$, are part of the same view, $v_1$, and a new view, $v_2$, without $n_2$ is delivered to $n_1$ then a view, $v_3$, without $n_1$ is delivered to $n_2$ before both $n_1$ and $n_2$ can share a view, $v_4$, again, where $v_4$ is installed after the other views and $v_2$ and $v_3$ are preceded by $v_1$.

pPRiDe is correct in static networks. The argument behind this is based on the correctness of PRiDe. PRiDe is assumed to be correct if the nodes within a view share a common start state and the view is static. A static view provides the properties necessary for PRiDe to work correctly. That is, reliable and ordered communication and a static and known node set. The communication layer excludes nodes that fail to deliver messages but since we only care about static views, so far, we ignore this case. pPRiDe works in the same way as PRiDe within a static partition. This is validated in section 6.2. The initial generation can be made stable when a common start state is assumed. The final generation is only used when the view has been superseded which does not happen if the network is static. Since the view do not change in a static network a generation is regarded as stable when it contains a delivered message from each of the nodes in the view. This is the same behaviour as when PRiDe is used in a static network consisting of the nodes in the partition. Given the correctness of PRiDe in static networks the correctness of pPRiDe in static networks follows.

pPRiDe is correct in networks with monotonically growing partitions, that is when nodes may not leave a partition but nodes may join. To show that this is correct we start by showing that generations in old partitions continue to stabilise correctly. Then we show that the final state of an old partition is shared amongst its members. Finally we show that all these final states are combined into a common start state for the members of the new partition.

Superseded partitions stabilise correctly. This argument extends the one for static networks. Assume that views $v_1, v_2, \cdots, v_n$ have merged into $v_{n+1}$. Assume that all nodes ignore nodes joining $v_{n+1}$ from other views than their prior view, that is filter out their messages. Then the old view can be regarded as unchanged, since all members are still reachable and the other nodes are ignored, and the argument for static views still holds. Messages in pPRiDe contain the unique id of the addressed view. Only nodes that once were members of a view can address that view and only messages addressed to a view can change generations in it. The id is used to discard messages addressed to unknown views to ensure that they cannot affect non-members. Old views can, in monotonically growing partitions, thus be regarded as static partitions for which correctness has already been shown.

Nodes that are members of a view that has been superseded, in monotonically growing partitions, eventual agree on common final states for the superseded views. In pPRiDe, nodes stop to append own updates to conflict set of old views. Since already installed views are static this means that the nodes eventually install a new view after the partition has changed and stayed stable. The conflict set of the old view is kept until all nodes from the old view, for example $v_1$, have installed $v_{n+1}$ and all updates made to $v_1$ has been stabilised. Since the old view can, for monotonically growing partition, be regarded as static, all nodes from, for example, $v_1$ stabilise all
generations in \( \nu \), to the same value. In monotonically growing partitions, all nodes install a common view since the membership service guarantees that nodes that stay connected eventually share view. Once a node has installed a new view it may not address own updates to the old, now superseded, view. This guarantees that all nodes, eventually, cease to address update to old views. PRiDe guarantees that all nodes eventually reach a common state if they all cease to update the database. This is also true for pPRiDe since the old view can, if the partition grows monotonically, be regarded as a static network. Since pPRiDe and PRiDe has the same behaviour in static network each node that shared the initial view eventually reach a common final state for this view. Assuming that the nodes agree on a common start state before updates to later views are applied this argument extends to succeeding views too.

The final generation in a view is only stable after a final and shared state for that view has been derived. When a node stabilises the last known generation it multicasts a finalisation message addressed to the final generation in the view. All other nodes with the addressed view installed have all the generations known to the stabilising node. Being stabilisable, in monotonically growing partition, implies that it contains one message from each node that are member of addressed view and thus eventually installs it. One node is always the last to send a message to the final generation of the view. The sending node knows about all generations it has initiated. For these generations to be stabilisable they also must be known by the other members of the view. Being the last node to send the finalisation message it also knows about all generations initiated by the other members. All nodes in the view are, if pPRiDe is correct in static networks, in the same state when the final generation is stabilised. At this point the stable state is sent in a initialisation message addressed to the initial generation in the superseding view.

All nodes that share a view eventually, in monotonically growing partitions, install a common initial state for the shared view. The initial state is deterministically derived from the initial generation once it is stable. For monotonically growing partitions the necessary state messages are delivered to all members of the view by the communication protocol. All nodes derive the initial state from an equal set of state messages resulting in equal initial states. This initial state ensures that all nodes reach the same state when they stabilise later generations. Old views are not updated after the initial state of the next view has been derived since the necessary initial messages are sent by the nodes after they have ceased to update and processed all possible generations in the previous view. Without any pending updates from earlier views this initial state, when installed, let us in the case of monotonically growing partition regard the view as a new static network. No updates addressed to the view are stabilised before the initial is derived ensuring that once they are processed they are applied to the same state on all nodes in the view.

We have shown that, for monotonically growing partition, old views can be regarded as static networks. Correctness for such partition was shown prior to this argument. The argument for static networks depended on a shared initial state, We have shown that such initial state is derived from the final state of earlier view. New views do thus have all properties necessary for the argument to hold for them to.
pPRiDe is correct when nodes leave a partition. The argument behind this is based on history branches. Updates performed by two nodes is said to belong to different history branches if the nodes may have issued the update concurrently while disconnected from each other. The view synchronous group communication protocol either can be delivered to all members of a view or the nodes that do not deliver the message are excluded. This is illustrated in figure 6.1 where the ellipses illustrate views, node $n_3$ is the only node that share history branch with node $n_4$ from time $t_1$ to $t_3$. Nodes with updates from different history branches eventually belong to different views. This allows different history branches to be held separated until their different states have been merged as part of an initial state transfer of a common view. If a view, $v_1$, splits into two or more new views, $v_2, v_3, \ldots, v_n$, then eventual consistency is guaranteed between all nodes that share new view. The view-synchronous group communication guarantee, by definition of agreement, that all nodes that share a series of views, $v_j, v_{j+1}, \ldots, v_k$ delivers the same set of messages in the same order between the installation of $v_j$ and $v_k$. pPRiDe filters out messages addressed to a view that are sent by a node that do not share history with the node from the target view until the current view. The required exclusion property guarantees that this filtering eventually is mutual and the uniqueness property ensures that the target view can be uniquely identified. This ensures that a propagated message only are processed by other nodes that have processed the same set of prior updates addressed to the target view as the sender of the message. Nodes that fail to deliver the message are excluded from the view by the communication layer and form their own view with a separate history branch possibly reaching a different state. If such history branches merge again then their different states are merged as part of the initial state transfer.

If a generation, $g$, associated with a view, $v_1$, becomes stabilisable for one node, $n$, in view $v_m$ then all nodes that share all views with $n$ in a sequence from $v_1, v_2$ until $v_{m+1}$ stabilise $g$ to the same value as $n$. For $g$ to be stabilisable, by definition, $n$ must first have delivered messages addressed to $g$ from all nodes in $v_1 \cap v_2 \cap \ldots \cap v_m$. A message addressed to $g$ is always delivered to a node after $v_1$ has been installed.
since it is associated with $v_i$ and the communication layer newer delivers messages before the view of the sender is installed at the receiver. The messages required for $g$ to be stabilisable must thus be delivered by $n$ after $v_i$ and before $v_{m+1}$. The agreement property of the communication protocol states that if one node delivers a message before a view then all members that share the view do so or are excluded. Thus all messages delivered by $n$ after $v_i$ and before $v_{m+1}$ must also be delivered after $v_i$ and before $v_{m+1}$ by all other nodes that share all these views. This shows that all nodes that share history from the view of $g$ can stabilise $g$ in the same view and the state is based on the same set of state messages. Nodes are thus either separated by a state transfer that merges divergent copies or share all updates messages that define the changes of the state of the object since the last state transfer.

We have shown that pPRiDe is correct in static partitions, in growing partitions and in shrinking partition. These are all possible ways for a network partition to change and pPRiDe can thus be assumed to work correctly networks with the targeted properties.

### 6.2 Validation

The main purpose of the simulator is to validate that the different nodes reach a common state if all nodes cease to update the database. This was tested by simulating nodes that frequently updated a colour value stored in the database. This colour value was visualised by colouring half of the range circle for each node with the stable value of the colour in the nodes database and the other to the tentative value (see figure 4,5). The frequency the node used the stabilisation function, the stabilisation frequency, was set low enough for each node in each network partition to get different stable value. All nodes ceased to make update to the database at a certain point in time but kept stabilising the database. When this happened all nodes within each partition, at the pace of stabilisation, reached a common intra partition stable colour. This behaviour does not only depend on the correctness of pPRiDe but also on the conflict routines for update conflicts to be deterministic.

The inter partition state was also observed during the validation. As a node migrates to a new partition so does the state of the colour. When the migrating node reaches a new partition all nodes of the new larger partition uses each node’s old stable state to derive a new common state. Whether the different partitions can reach a common global stable state without forming an unpartitioned network depends on the state conflict resolution strategy. We use a number value instead of a colour to illustrate this. If the mean was used to resolve state conflicts then the partitions would not be guaranteed to reach a common state if one node travelled from one partition to another and then back. For example, two partition, $p_1$ and $p_2$, of five nodes each $p_1$ start with a stable value of 1 and $p_2$ 10. If one node leaves $p_1$ and connects to $p_2$ then the state would be resolved to $(5 \times 10 + 1)/6 = 8.5$. Later the node leaves $p_2$ and reconnects to $p_1$ and the state is resolved to $(5 \times 1 + 8.5)/5 = 2.7$. In contrast, if they used the maximum values to resolve state conflicts both $p_1$ and $p_2$ would have reached the same state. This behaviour can be observed visually in
the simulator.

No timing measures were taken during the evaluation. This is because the simulator does not simulate network delays and transmission failures. Without these aspects timing measures would be of little use.

![Graph showing the length of the conflict set sampled every minute](image)

**Figure 6.2:** The length of the conflict set sampled every minute

The visual evaluation does not show if new updates eventually are stabilised and integrated into the database. But if no updates are removed from the conflict set unless they are integrated in the database then the conflict sets would grow over time if some updates newer was integrated. A code walk through shows that no updates are removed unless they are integrated into the database. The size of the conflict sets can thus give an indication of whether all updates are integrated. To evaluate if this happened the average size of the conflict set over the last minute of 10 nodes was recorded every minute for 1000 minutes. The ten nodes formed a network with an average partition size of 2,65 nodes and the partitions changed on average every 12,6 second. The database stored a single object, a colour, in the database. The hue of the colour was update randomly based on a normal distribution with a mean of 1 second and a variance of 0,3. The stabilisation was also performed based on a normally distributed interval with a mean of one stabilisation every 10 seconds and a variance of 0.1. The result of 10 runs of this experiment is plotted as a run chart in figure 6.2. The different quartiles and the median is plotted as a separate line. The graph indicates no tendency for the conflict set to grow larger over time.

The dynamics of the nodes can explain the peaks in the graph. If nine nodes in the experiment share the same partition then each node receives updates from each of the nine nodes. Each node use an update frequency of 1 update per second.
each node receives updates from each node in the partition, that is 9 updates per second. A node will thus, on average, receive 90 updates between each stabilisation. If the last node in the simulation joins the partition then each node creates a new conflict set from which no updates can be pruned before all nodes has stabilised the last update in the old conflict set. If the last node to stabilise does so after 10 second then waiting for this to happen, on average, 100 updates has been received and added to the new conflict set by each node. If it takes 10 more seconds before the last node has stabilised after the integration of the new conflict set then it can have received 100 more updates resulting in a length of 200 non integrated updates in its log.

This growth, which depends on the behaviour of other nodes and size of the partition, can result in unpredictable resource usage. This might seem to contradict the goal of predictable resource usage but it is not dependant on the lifetime of the partition and can be controlled. A node can voluntarily leave the partition and as a single node partition stabilise all data independent of any other node. This is described more thoroughly in section 7.4.1. pPRiDe is designed to have the same behaviour as PRiDe in networks where both can be used. To validate this, the simulator was adapted so that all nodes held two databases. One with PRiDe replication and the other with pPRiDe replication each storing a colour object. The update and stabilisation frequency was unchanged from the previous experiment. But each time one database was stabilised or updated so was the other. For PRiDe to work the network topology had to be static and without partitions. The nodes updated the database for the first 2 minutes and then ceased to make new updates but kept stabilising. This evaluation was performed ten times. First the nodes had one half coloured with the tentative value from PRiDe and the other half from pPRiDe. For the second evaluation the halves were coloured by the stable value. No difference between the halves could be seen during either of the evaluations and all nodes reached a common state after the updates ceased. Also the size differences between the conflict sets of the two protocols were recorded during the experiment showing that the size was equal during the simulation.

These three simulations support, by exhibiting an expected behaviour, the correctness argument in section 4.2.1. First, the nodes in a partition reach a common state. Second, new state information from other partitions get integrated when it becomes available. Third, the intra partition state is equal to that of PRiDe that is assumed to be correct.

### 6.3 Comparison with existing protocols

In this section pPRiDe is compared to other replication protocols. First it is compared to PRiDe to highlight what differences the proposed extension in pPRiDe introduces to the base protocol. Later pPRiDe is compared to two other protocols, Bayou and Bengal, which are also designed for nomadic computing with frequent partitions.
6.3.1 PRiDe

In networks with the properties necessary for PRiDe to work pPRiDe can be used without view-synchronous group communication. Since PRiDe assumes that no messages are lost and received in the same order as they were sent, pPRiDe can mark received messages as delivered immediately upon reception. The overhead of using pPRiDe in such situation is the space necessary to store the Boolean value that marks the updates as delivered. PRiDe also assumes that no network partitions or membership changes occur, pPRiDe do not perform any state transfers in such networks. The communication between the nodes should be the same for both protocols but pPRiDe uses slightly larger messages since it adds identifier for the view and the id of the sender. The changed definition of when an update is regarded as stable is only different from the original definition in cases when there are network partitions and is thus equal in networks where both can be used.

6.3.2 Bayou

Bayou (Terry et al. 1995) is designed for collaboration in situations where the different collaborators have little need for frequent synchronisation (Edwards et al. 1997). pPRiDe, in contrast, is designed for situations where different collaborators cannot rely on frequent synchronisation but it is beneficial to take advantage of it when it is possible. Infrequent updates may result in lots of conflicts that should be resolved once the node synchronises. Both Bayou and pPRiDe are non-transparent and rely on the user to supply policies for how conflicts should be resolved.

Bayou like pPRiDe uses operational transformation to resolve conflicts. But Bayou uses what is called an anti-entropy strategy where synchronisation is performed in pairs. In contrast, pPRiDe uses a per partition synchronisation strategy.

Both protocols primarily use update transfer but resorts to state transfer to manage the length of the log of updates. pPRiDe is designed with short response times in mind and frequently merges updates to keep the log short, as a result, state conflicts are common. This is solved by application specific policy in a way similar to the one for update conflicts. In Bayou, to my best understanding of Petersen et al. (1997), state conflicts are resolved by using one of the states as a base-line in the pair wise synchronisation and the other is discarded possibly discarding important updates.

6.3.3 Bengal

Bengal (Ekenstam et al. 2001) is a pure state transfer protocol. It uses version vectors to detect if the state of replicas conflict with each other. This is the same strategy proposed in section 4.2.1 as a way to reduce the number of state conflicts in pPRiDe. Bengal and pPRiDe share most properties if they are used in situations where partitions larger than one node is infrequent and nodes only rendezvous to synchronise. But the use of operation transfer in pPRiDe can be beneficial when
the partitions gets larger and synchronisation more frequent. This comes at the cost of having to provide conflict routines for both state and operation conflict for pPRiDe. But can also reduce the number of conflicts when for example two operations commute.
Conclusions

This chapter summarises the contributions of this thesis, presents future works and concludes the thesis with a discussion.

7.1 Summary

Partition aware data replication is a suitable method for data sharing in mobile ad-hoc networks. In such networks the lifetime of network partitions cannot be controlled unless the mobility of the nodes is limited. Replication protocols used in such networks do thus need to handle partitions with unlimited lifetime. Existing replication protocols designed for disconnected work handles single node partitions with long lifetime but lacks support for larger partitions. Other replication protocols handle larger partitions but are designed for systems where partitions are short term failure modes.

An existing database replication protocol, PRiDe, can be extended to use a partition aware view-synchronous group communication. This communication protocol provide both a partition wide multicast primitive with delivery guarantees but also information about what nodes that are members of the partition. Data changes are propagated as updates, as opposed to states, between the members of a partition. State messages are used to derive a common baseline to append updates to when the partition configuration change.

The presented replication protocol, pPRiDe, uses application specific forward conflict resolution. Frequent updates to the same object in two or more Long lived partitions may result a large set of conflicts. Backward conflict resolution may in such cases result in very old updates being reverted, removing updates that change the data to reflect the current state. Reapplying such updates can create an unpredictable workload for the application program. Forward conflict resolution transforms conflicting updates/state into conflict free new ones. How to perform such transformation in a sensible way is application specific. To make pPRiDe general, user specified resolution routines are used.
7.2 Contributions

We have provided one piece of the puzzle necessary to realise the emergency response system described in the introduction. The pPRiDe method presented in this thesis shows that it is possible to use a database with fully replicated data in a weakly connected ad-hoc network. This comes at the cost of relaxing the data consistency guarantees between temporary network partitions.

The main contribution of this thesis is pPRiDe, a version of the PRiDe replication protocol extended with support for long-lasting network partitions. The resource usage is independent of the behaviour of nodes outside the own partition. In addition to the main contribution, this thesis also contributes an implementation of PRiDe built as a separate module on top of a database and a simulation and database platform suitable for future research.

7.2.1 pPRiDe

pPRiDe is the main contribution in this thesis. pPRiDe is an extended version of PRiDe that adds support for arbitrarily long lived partitions while not doing away with the predictable resource usage of PRiDe. pPRiDe adds state transfer to integrate nodes into a network partition to the otherwise purely operation transferring PRiDe. PRiDe, in contrast to other replication protocols (Edwards et al. 1997, Ekenstam et al. 2001, Kermarrec et al. 2001), resolves conflicts between sets of conflicting updates instead of pairwise resolution. pPRiDe extends this approach to the resolution of conflicting states for the added state transfers. This can reduce the number of states visible to the application program in, for example, a three way merge the conflict is resolved with one state transformation instead of two.

Arbitrarily long lived partitions are a problem for update transfer protocols but not for state transfer protocols. Update transfer protocols must store all updates performed since the last globally common state and with long lived partitions this introduces unbounded resource usage not acceptable for real-time systems. State transfer protocols use unnecessary amounts of bandwidth in situations with frequent synchronisation and reduce the information available at conflict resolution. pPRiDe uses a hybrid strategy at the cost of requiring application specific conflict resolution functions for both state and update conflicts. Update transfers are used between nodes in the same partition and state transfer is used to merge the state in different partitions that merge.

7.2.2 PRiDe implementation

There exist no implementations of PRiDe prior the one presented in this thesis that uses an existing database. Holmgren (2006) presented two strategies for how PRiDe could be implemented on top of an existing database but did not implement any of them. The implementation presented in this thesis is based on the presented strategy where the replication is built as a module between the application and the
The existence of such implementation clearly shows that the proposed strategy is feasible.

### 7.2.3 Simulation platform

This thesis also contributes a platform suitable for future work in distributed databases in mobile networks. Each component in the system is loosely coupled to the other and can be exchanged to test new strategies. This platform can, for example, be used to implement other replication protocols or a different database engine. A database engine designed to store data with spatial extents as key is currently in the works.

### 7.3 Discussion

pPRiDe can be used to replicate real-time databases in ad-hoc wireless networks. Such replicated database can be used to facilitate data sharing and relieve the application programmer from some aspects of the unpredictable network. However, the replication protocol of the database cannot transparently hide all complexity. There is always a risk of update conflicts in networks with network partitions if multiple nodes are allowed to modify the same objects. The forward conflict resolution used in pPRiDe requires data specific resolution functions for both state and update conflicts. Adding these is probably less work than developing a complete application specific data sharing strategy but it is still a complex task.

It is possible to use pPRiDe right now, but before it is possible to depend on it an evaluation in a real environment is necessary. In this thesis we have assumed that view-synchronous group communication does not introduce unbounded resource usage when used in mobile ad-hoc networks. Further more we have assumed that the unique and exclusion properties described in section 6.1 are implementable. We do not foresee any problem implementing them but since we lack a working design we cannot guarantee that it is possible.

Bandwidth usage is thought to be a major issue for deployment. pPRiDe can be optimised to reduce the bandwidth usage in some situations (see section 7.4). This is mostly a problem when large objects are stored in the database. This is however also a situation where pPRiDe has the possibility to use less bandwidth than pure state transfer protocols. This is one of the big benefits of the more complex hybrid strategy used in pPRiDe.

We would also like to improve the stringency of the pseudo code presentation of pPRiDe. We believe that the code can be simplified further and be made more consistent. For example, the representation of the initial and final generations should not require a special representation in comparison to the ordinary generations.
7.4 Future work

In this section we give a brief introduction to areas from which pPRiDe could benefit from more work.

7.4.1 Voluntary partitions

The protocol must be extended to gracefully handle overload situations if it is to be used in unconstrained mobile networks with real-time requirements. Messages from other nodes are one source that can cause such overloads, both from processing and storage of undelivered and unstabilised updates. Syberfeldt (2007) proposed the use of voluntary partitions as a possible means to solve such situations in PRiDe. pPRiDe provides the support for partitions necessary for this but provides no policy for the creation of voluntary partitions. This policy must decide when to form partitions, where in the current partition to split and when to merge the partitions again. For example, a node that temporarily requires hard-real time guarantees and for which missed deadlines are critical, can create a single node partition to reduce the unpredictable load from other nodes. In a soft real-time where missed deadlines only decrease the value of the result, the policy can be less restrictive.

7.4.2 Communication overhead

The communication overhead has not been regarded when PRiDe(Syberfeldt 2007) and pPRiDe was developed. Reduced communication overhead is especially important in battery powered mobile systems where reduced communication can result in lower requirements for battery capacity. Both the number of state transfer messages and the number of messages to inform that no updates have been performed can be reduced. How the latter can be realised is described in Syberfeldt (2007). These are directed to PRiDe but should also be applicable to pPRiDe.

The number of transferred states can be reduced in three circumstances. First, all members of a new view send one message containing their final state from the previous view. Of these message all sent from nodes that shared the previous view contain the same state. It should thus be possible to reduce the number of such messages so a node only receives a state message from one member of each of the now merged views. Second, state transfer is performed every time a view changes, even when all nodes in the new view also were members of the old. This is obviously not necessary and adds unnecessary state transfers every time a node is excluded from a view. Third, pPRiDe does not check if the version of the object replicas differ between two merged views before a state transfer is performed. If, for example, one node leave the partition and later rejoin it then the state of each object replica are exchanged between the nodes even if the state is unchanged since the node left the partition. Version vectors (see section 2.5) could be used to solve this. Each object has a version vector that is compared with one of the merged partition and the state is only exchanged if they are different.
7.4.3 Storage overhead

The storage overhead can be reduced. Both the communication protocol and pPRiDe separately, upon reception, store the message data. The communication layer stores the data until it is possible to deliver the message and pPRiDe stores the data until the generation is stabilised. Since the stabilisation depends on the delivery it always happens after the delivery. This should make it possible to share the data structure between the communication protocol and the replication protocol and in that way reduce the resource usage.

7.4.4 One way communication

One way communication modes is used to improve battery life times in mobile communication systems (Fife & Gruenwald 2003). It would be interesting to extend pPRiDe to allow nodes with low power to integrate update messages from other nodes but not send its update to the others. This would of course result in a divergent state as soon as the node performs an update, but the node can at least act upon more up to date data than if it ignores the information intercepted from the other node.

7.4.5 Transactions

The current implementation lacks support for transactions. PRiDe has some support for transactions. We see no reason for why the approach used in PRiDe should not work in pPRiDe but this must be investigated more thoroughly.

7.4.6 Adding and removing objects

pPRiDe assumes that the set of objects is static, that is no object can be added or removed. This ensures that the number of objects is bounded and predictable but can often be too restrictive.

Adding objects can be implemented with a new message type that transfers the initial state and the key of the object to the other nodes. Removing objects is more complex since it has side effects. If for example an object is removed, then it should not be reintroduced when the node synchronises with another node that do not know about the removal. However, if a client program explicitly reintroduces the object then the synchronising node should also reintroduce the object. One way to solve this problem is to use tombstones (Saito & Shapiro 2005). An object that is removed is replaced with a tombstone.

7.4.7 "Fuzzy" keys

It would be interesting to evaluate how pPRiDe can extended to support user provided policies for when to merge observations of the same phenomena stored
with slightly different keys.

Mobile database nodes are useful because they can be carried by a person who collects observations about the surroundings. The coordinates, reported by a GPS, can be a candidate database key for such observations, letting other nodes query for nearby observations. In the weakly connected ad-hoc network many node may store the same observation independent of each other. When multiple such observations reach a common node they ought to be merged, but since the GPS coordinates are not absolutely accurate multiple observations of the same object may have slightly different keys and different object may have equal keys. When to merge multiple observations of what might be the same object is studied in information fusion where it is part of the level one, object refinement, in the JDL model (Llinas et al. 2004).
Bibliography


