An Exchanging Algorithm for Database Replication Protocols

A middleware metaprotocol for multiple concurrent protocols

F. Castro-Company and F. D. Muñoz-Escoí

Technical Report ITI-ITE-07/02

January 25, 2007

Abstract

Database replication task is accomplished with the aid of consistency protocols. This work starts addressing the study of a metaprotocol that manages consistency protocol exchanges. This is accomplished by means of a study of compatibility among several consistency protocols that may be working concurrently and with a framework that allows multiple protocols execution, which completes the objective of our study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

During recent years a series of replication protocols have appeared in order to fullfill the task of consistency management in replicated systems. These protocols can be classified according to several parameters (architecture, interaction, termination [25]) and they can be used with applications that deal with different database transaction isolation levels (serializable, read committed, snapshot).

We address here a wrapping task: We want to describe a metaprotocol capable to provide us with administration operations over protocols. These administration operations are: stop protocol, start protocol and change protocol.

From now on, we'll use the following definition for a metaprotocol:

Definition: Metaprotocol

A metaprotocol is an algorithm that allows several consistency protocols to be executed over the same database.

A metaprotocol allows the system to execute two concurrent transactions that use two different replication protocols; it allows a transaction that was started using a protocol to continue its execution using another protocol; it allows the system to finish a given protocol waiting for all started transactions to stop and to start a new protocol letting all new transactions to use it. The first example refers to concurrent protocols execution. The two next ones refer to a protocol change but they are tightly related to the concurrent protocols execution case because they allow protocol changes not to be necessarily executed sequentially.

Our goal is to obtain a light and efficient metaprotocol that achieves these tasks with minimum functional impact for the users. As most consistency protocol advantages and weaknesses depend on variable parameters ([17], [13]), such a metaprotocol is an essential piece to turn a distributed database system into an adaptable one.

This goal's scope is general but we can easily justify all efforts taken in this direction it if we aim towards complex and large systems:

It is not unusual for a large organisation to use an information system[1] that accesses common database schemas.

[1]For us it is indifferent whether the system is composed of stand-alone applications that cover special business needs or it is an enterprise suite.

Good examples of common schemas are those regarding to staff information or corporate resources. Staff information can be used for administrative tasks as well as for login and profiling and access purposes. A corporate resources catalog goal is to centralize information about headquarters, office branches, professional categories and all sorts of business related information classified by the organisation. While the administrative personnel will need the latest versions or exclusive accesses, other employees and external users will need a less accurate access to data.

The first elements a developer finds to overcome this scenario are transaction isolation levels. All competitive database management systems currently offer isolation levels. On the applications layer *JDBC*, for example, allows connections for Java applications to use transactional database capabilities.
Transactions and isolation guarantees are applicable to a single database server however, as organisations grow, the information system infrastructure has to scale further. The first steps taken to improve scalability usually consist of adding CPU and memory power to the server but soon we realize that this is not enough. Growth implies an increase of the number of users thus an increase of the number of offices accessing the server in a way that finally leads to the use of replication.
Replication can be embedded either in the underlying database management system or in a middleware layer between applications and the database. Whatever approach is taken the replication system has to allow transaction isolation guarantees and, frequently, variations (see [7]) of the ANSI ones (see [28]) are more convenient.
Imagine the case of data marts or data warehouses; other examples are monthly payroll calculations and closing of financial years. While the data warehouses mostly need guarantees similar to those offered by control version systems all the time, the last examples need much more restrictive guarantees during a certain time intervals but not always.

As we will cite and display later there exist different architectures and plenty of replication protocols in the literature. As each architecture displays advantages and disadvantages when trying to solve any of the above concerns, recent studies such as [19] and [27] compare protocols performance. [19] conclusions expose ROWAA approach as the most suitable for the general case and [27] is based on total order broadcast techniques.
While agreeing with these results, regardless of any predictable improvement on network bandwith we find that there will always be a lot of scenarios such as the two examples depicted before that benefit from lazy and epidemic approaches. Obviously ROWAA and total order broadcasts would update steadily in all nodes a change performed over a table of streets and cities in a corporative resource catalog but a deferred update would imply no functional burden and this update would cause much less interferences in the performance of other applications.

So, instead of proposing a consistency protocol with general purposes, we have designed a metaprotocol that allows a representative set of consistency protocols to work concurrently. This metaprotocol is a solution to the above concerns as applications will be able to take advantage of the protocol that better suits their needs. Not only the best protocol for the general case can be selected for an application but also it can be changed for another one if the application access pattern changes drastically or if the system overall performance changes due to specific load variations or network or infrastructure migrations. These selections and changes can be forced by an administrator following theoretical and empirical studies or they can be automatically triggered according to background performance analysis[2].

The paper is organized as follows. In section 1.2 the general scenario is presented. In section 1.3 we list a set of desired capabilities and we make some considerations about issues related to behaviour for the

[2]Selection issues are out of the scope of this study.

essential pieces of the system.
At this point the work follows two parallel paths that lead to the metaprotocol architecture.
One of them, described in chapter 2, presents the metaprotocol administrative operations (stop, start and change) that allow changes in the working state of the protocols. In order to approach the problem gradually, firstly the state changes are considered sequential (see appendix A) and later they are allowed to be concurrent.
The remaining path of the study is shown in chapter 3. Here we perform a study about the metadata involved in protocol changes. A set of different protocols with different characteristics is collected and then each protocol information about metadata is used to find the characteristics they all have in common.
Once this is done, operations and data (metadata) are combined to achieve concurrency effectively. This is described by means of an architecture (including metadata management and execution control) for the metaprotocol existence shown in chapter 4.
Chapter 5 summarizes results and contributions and inquires further into the motivations of this research.

1.2 Scenario

Our scenario is composed of a set of *nodes* n_i (sometimes called servers) such that $i \in [1, N]$. User applications do not necessarily exist in these nodes[3] but they access the databases and possibly certain server services in those nodes. Users as well as applications are referred to simply as *users*. Each user accesses one of the nodes (always the same and typically the closest one) so that each node has a given subset of the users accessing certain data.

Data is composed of a set of *objects* ($o_j,\ j \geq 1$). It is stored in the nodes' databases (also called *repositories*) and it is replicated through the set of nodes completely or partially. In partial replication, the set of objects is different depending on the node or group of nodes: changes are sent to a subset of nodes. In total replication changes are sent everywhere.

Objects are replicated in order to optimize accesses and data consistency is achieved with two means: database transactional capabilities and consistency protocols.

Accesses are organized in transactions ($T_{ik},\ i \in [1, N], k \geq 1$). Transactions group a set of operations as an atomic step in a way that all operations are applied (commit) or they are all aborted (rollback). Each transaction is started by an user in one of the system nodes and it is described with its readset (rset) and writeset (wset).

Consistency protocols ($P_m,\ m \geq 1$) replicate by means of communication protocols and guarantee consistency by means of consistency logic plus recovery protocols. They expand transactions through the system in order to coordinate actions in each node to provide the required transaction isolation levels. These levels are based on [10] and [7].
So, while consistency protocols are used to make transactional properties prevail through the system, communication protocols take care of message delivery amongst nodes and recovery protocols are in charge of maintenance during node crashes. All of them rely on membership protocols in order to add awareness of node crashes and node joinings.

It is worth saying that consistency and recovery logic are tightly coupled. In fact, many times both tasks are considered and mentioned simply as *consistency tasks* because recovery logic is usually highly

[3] See, for example, downloaded *Java WebStart* rich clients.

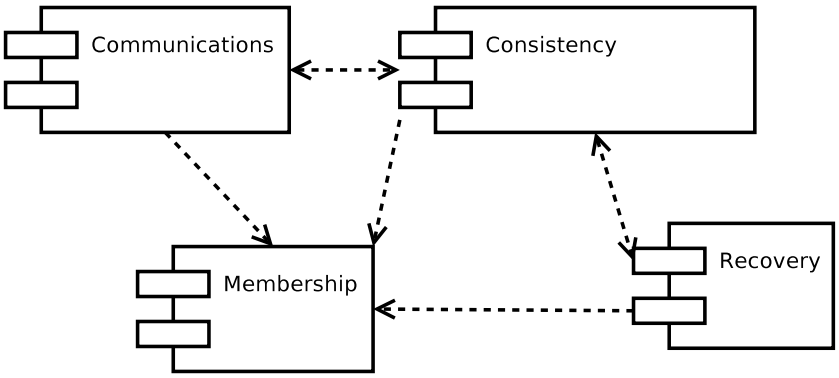

Figure 1.1: Consistency Components.

embedded inside the consistency logic. If nothing else is said, in our study we will also exploit this idea whenever we mention consistency protocols.

All these components (see figure 1.1) are present in each system node.

In order to have a general idea of what kind of information a consistency protocol manages, let us say that some make use of the concept of ownership: even though the object is accessible everywhere, each object belongs to, or it is controlled by, a certain node[4]. Notice that this is not necessary for all protocols: Other protocols use communication primitives that prevent nodes from being managers of a certain set of objects. In any case, consistency protocols manage additional information outside of the scope of the users information. This information is called *metadata*:

Definition: Consistency protocol metadata

Metadata is additional information (persistent or transient) associated with each object or transaction.

This information can be versions, timestamps, transactional context and the like.

Protocols, objects and transactions model general information and processing. From a strictly practical point of view, nodes are usually servers located at different data centers. The kind of users and the information stored in the databases depend on the business the information system exploits. Applications range from ATM machines to online interconnected shops, health-care systems for patients clinical history management, logistical administration of stock stores, etc.

1.3 Framework

Initially, we assume we have a system for data replication (such as *GlobData* [2] or *Madis* [4]) prepared to hold a consistency protocol per data repository.

This system will provide an interface to the user in order to make replication as transparent as possible. These user calls will be captured by some means and then consistency tasks will be performed.

In these systems, allowing different sets of data to be accessed with different protocols is straightforward because each repository is associated to a protocol. For a single repository, if we want to change a protocol for another one, or better if we want to allow protocols concurrency we'll need additional components and

[4] i.e. the one where a user requested its creation.

an additional architecture wrapping the original one to allow such data exploitation.

Recovery logic is suited to each consistency protocol and we'll consider it as a portion of the consistency logic. Other parts, administration oriented, can be considered as common and will be the core of the metaprotocol architecture.

1.3.1 Communications

As communications logic is usually common, communication protocols such as [1], [3] or [5] offer a set of general communication primitives (unicast, broadcast or multicast) that provide certain message delivery guarantees (reliability, atomicity, uniformity) [14].

We consider the communications suite as a black box as we find that the currently available suites contain operations to fullfill most consistency protocols requirements.
Between the chosen communications suite and the protocols it is necessary to add a new component that distributes network messages to their appropriate consistency protocol destination.

Protocol messages have to be wrapped inside general messages that carry information about the consistency protocol context.

Protocol ID	Message Type	Message

Once the message is sent to the *CommDistributor*, it unpacks it and it is delivered to the corresponding protocol (See figure 1.2).

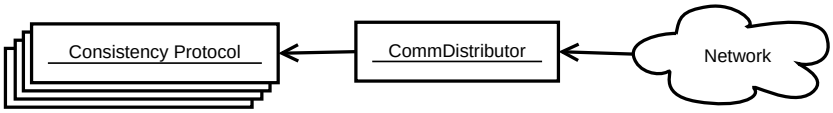

Figure 1.2: Message distributor.

Furthermore, for the sake of efficiency a good choice for this common API will include a communications suite that is able to group the messages that need to be ordered from the ones that need not to. For example: no ordering is required between messages from different protocols and ordering is required between metaprotocol administrative messages and the rest of them.

1.3.2 Concurrent protocols

Protocol concurrency motivations were explained in the introduction. Typically different applications accessing the same data repository are able to benefit from the different protocols guarantees. One important subtask of a multi-protocol environment is to allow an application to change the consistency protocol it uses. A protocol change may be necessary due to several reasons: A monitoring system detects that the response times and the abort rates are not adequate; a given application knows for sure which protocol is more suitable for its needs; a given transaction access pattern is best served with a certain protocol.

There are several ways to change a protocol (say P_1) with another one (say P_2): sequentially and in parallel.

Sequential protocol exchange

This means that P_1 is stopped in all the nodes and then P_2 is started.

Definition: Sequential protocol exchange

For P_2 transactions (t_{2i}) to start, all P_1 transactions (t_{1j}) must have had finished so that no t_{2i} and t_{1j} are concurrent.

During P_1's stopping, something has to be done with user calls to create new transactions:

- **Make them wait**: Create transaction calls are blocked until the protocol is stopped.
- **Make them abort**: Create transaction calls throw an exception.

The second choice is suitable to completely stop the system (i.e. when no other protocol is intended to be loaded) and the users are reluctant to stop their accesses.

The first choice needs to know when the protocol is stopped. A protocol is stopped when no local[5] or remote[6] transactions exist in execution. As soon as this happens, create transaction calls can be unblocked.

This can be achieved with a transaction counter. For the counter to work, protocols must inform the MetaProtocol about the creation and finalization of transactions. As soon as a node counter reaches zero the rest of the nodes should be informed. See figure 1.3 for an initial draft of the architecture design.

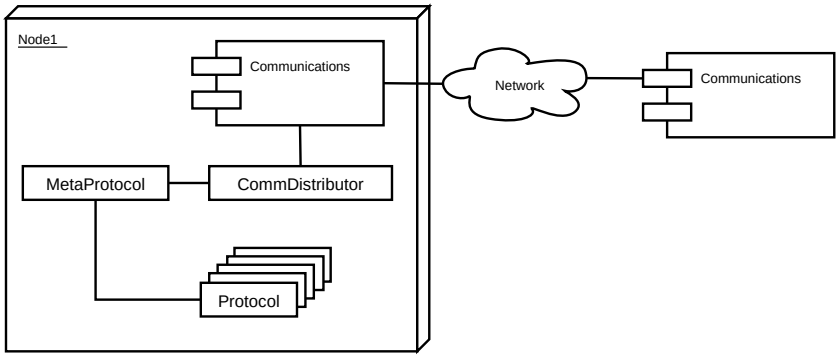

Figure 1.3: Metaprotocol.

Parallel protocol exchange

An ideal parallel changing implies not blocking the create transaction calls while the previous protocol is still running and being able to turn a t_1 transaction into a t_2 one. This change has to be transparent to the user and then, the object they obtain to perform operations is to be a proxy of the real protocol that distributes user calls to the appropriate protocol after performing control tasks.

When no change is being performed, the proxy simply redirects. When the changing is taking place it distributes calls to their corresponding protocol:

[5] A local transaction is a transaction whose create transaction call was invoked in the local node.

[6] A remote transaction in node n_i was created in another node $n_j, j \neq i$ and has sent a message that has already been delivered in node i.

- New transactions use protocol P_2.
- Already created transactions can take one of the following two approaches:
  a) Use protocol P_1: Sooner or later all remaining P_1 transactions would finish and the change would be completed.
  b) Change from P_1 to P_2 as soon as they can: This option helps the change to take place earlier.

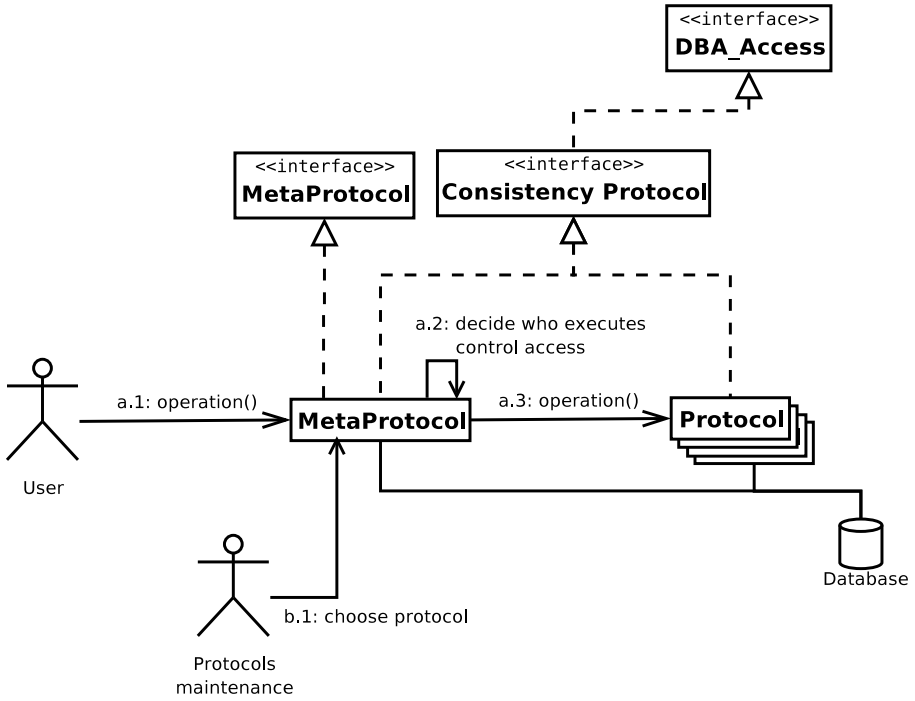

Figure 1.4: Framework for a general changing.

Figure 1.4 shows a more detailed architecture design to model these characteristics. For parallel protocol exchange, the same considerations about transaction counters done for sequential exchanges apply. Notice that once this objective is accomplished, the general task of concurrent protocol execution (without exchange purposes) will already be achieved.

Special care must be taken with different protocols and different database transactional isolation levels. The changing protocols metaprotocol has to guarantee that the safety properties of the individual protocols hold:

Definition: Safety extension for the metaprotocol

Safety properties for all individual protocols that the metaprotocol covers hold.

General changing of protocols

Sequential change of protocols can be seen as a particular case of parallel change. Both sequential and parallel changes need to define steps (there may be several) during the execution of transactions when the metaprotocol is able to request a protocol change. We call these steps *Change Points*:

Definition: Change Point

A change point is the step during the transaction execution when a consistency protocol change can be carried out. This means either that P_1 changes to P_2 or that t_1 and t_2 transactions coexist.

For the sequential case the *Change Points* are the beginning and the end of the transaction. As soon as we reach these points, the metaprotocol will decide whether to change (there are no other P_1 transactions) or to keep the new transactions waiting.

For the parallel case the *Change Points* can be any of the user calls (not only begin and commit/abort) or internal steps of the execution of those. It will depend on the involved consistency protocols characteristics.

1.3.3 Transactions and Sessions

Some systems such as [2] make a difference between transactions and sessions. Sessions contain several transactions. While in a system of such characteristics transactions can be committed or rollbacked, sessions can be created or closed. In fact, a session is like a *JDBC* or *ODBC* connection.

In this situation we can count either sessions or transactions. The most straightforward approach is simply to count transactions. If we choose to count sessions then new transactions will be created during a protocol change. This fact can be softened making a second count with the transaction information. A new transaction is created as soon as the previous one has finished with a commit or a rollback statement.

Liveness

For the exchange operations the liveness of the metaprotocol depends totally on the ability to bring the number of transactions/sessions of the running protocol to zero and being able to perform operations with the new protocol as soon as possible. For the protocols concurrency environment, the liveness of the metaprotocol lies on the ability to keep the individual protocols liveness by keeping protocols from voiding each other liveness properties.

In some exchange scenarios it may be usual to have transactions opened for a very long period of time while the user is performing scarcely any operations. These idle transactions will delay the actual exchange of protocols. The only way to skip this situation locally is to establish timeouts and measure time intervals between user accesses while protocol changes. If they surpass a certain threshold, transaction could be stopped.
Using distributed information, as soon as the percentage of nodes that are ready to perform the change has surpassed a certain amount, the local timeouts set for the local idleness detection should be decremented or set to zero to force the remaining transactions to finish.

If the main purpose is to perform the fastest protocol exchange then aborting working transactions is the most appropriate solution. However for most systems this solution is unacceptable. An empirical study can be performed for any desired scenario in order to optimally adjust the aborting thresholds but the best tunning needs human interaction in order to point out the current scenario, and furthermore we must expect this is a constant one.

In any case, notice that if we admit concurrent protocols execution none of these considerations are necessary.

We will consider all these facts external to the liveness[7] of the metaprotocol because they depend on the user usage of the system. So, regardless of the use of any approach to avoid long idle transactions or the use of none, we assume that transaction counters always reach zero in all nodes.

Definition: Liveness extension for the metaprotocol

The following properties must hold:

1. *Liveness properties for all individual protocols that the metaprotocol covers hold.*
2. *When requested, all these individual protocols eventually finish executing transactions.*
3. *Any protocol change or stop eventually finishes.*

1.3.4 Conflicts detection

To detect access conflicts locks or timestamps can be used but, in any case, these conflict solving operations depend on constant comparison of accessed objects sets.
Say transaction T_1 reads o_1, o_2 and writes o_1. At the same time, transaction T_2 reads o_1, o_3 and writes o_1. If both try to commit, there's a write–write conflict that must be solved. If fact, the latest of both write operations will be blocked waiting for the first one resolution (commit or rollback).

Care must be taken in order to avoid deadlocks: t_{1i} reads o_1, o_2 and t_{2j} reads them too. Then t_{1i} writes o_1, t_{2j} writes o_2, t_{1i} writes o_2 (and gets blocked) and t_{2j} writes o_1 (and gets blocked too).

Fortunately relational databases solve these situations by themselves but still, as we are adding network concurrency, consistency protocols have to be prepared to solve situations like these.

Furthermore, if we consider that different users might access the same data using different consistency protocols, conflict detection between different protocols has to be performed too. Figure 1.5 shows an example. In this figure P_2 suffers from a blocking of t_{22}. This conflict is detected with the regular consistency protocol.

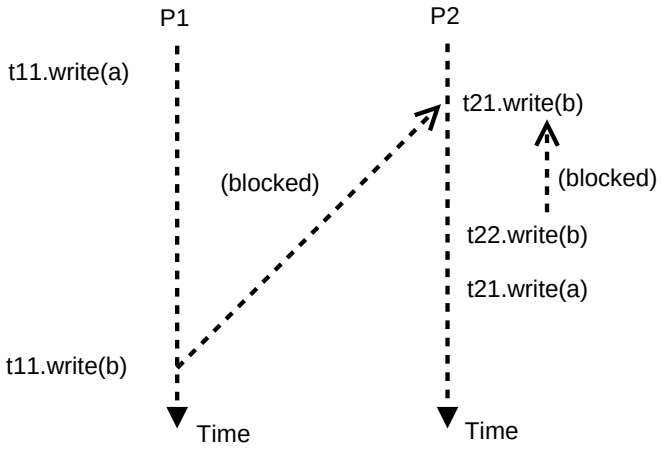

Figure 1.5: Conflict between protocols.

[7] For a formal study on how safety and liveness probe invariance and well-foundedness of algorithms, check [9].

t_{11} accessed through P_1 is blocked due to conflicts with P_2 accesses.

We will see in section 3.2 that protocols can be split into different general phases. Protocols offer an API to the users and we must identify which operations imply access to objects. These operations usually return or receive object result sets.

Definition: Result Set

A Result Set is the set of objects accessed during a transaction. It is decomposed into two different parts: The WriteSet (wset) for the write accesses and the ReadSet (rset) for the read accesses.
We assume the most general definition of Result Set (in order to contemplate all possible consistency protocols), whose compilation is considered to be incremental.

This indicates that the MetaProtocol should act as a common conflict detection component with methods like these:

```
public void accessObjects(TransactionMetaData tmd, ResultSet rs)
  throws ConflictException;

public void releaseObjects(TransactionMetaData tmd, ResultSet rs);
```

Using tmd the transaction id, its protocol and the current protocol phase can be obtained. These values identify which protocol and during which transaction execution phase accessed the objects result set (rs). Conflict detection is based upon comparison of sets of objects. A transaction accesses an object "for a while" and then it stops accessing it. This does not necessarily mean that we have placed a lock over the set of objects because some protocols declare the intention to access before the access is performed.
The methods signatures plainly say that the rules to decide about object access conflicts depend on the access type a transaction is using, the protocol it is using and the phase the transaction is at when the access is performed.
To show these rules, we need a previous study about protocols and their metadata (section 3.2) and that is why they are introduced later in section 4.4.

As protocols are actually behind a proxy, before or after the call to the real protocol, a call to the MetaProtocol can be done to check the conflict situations (see figure 1.6).

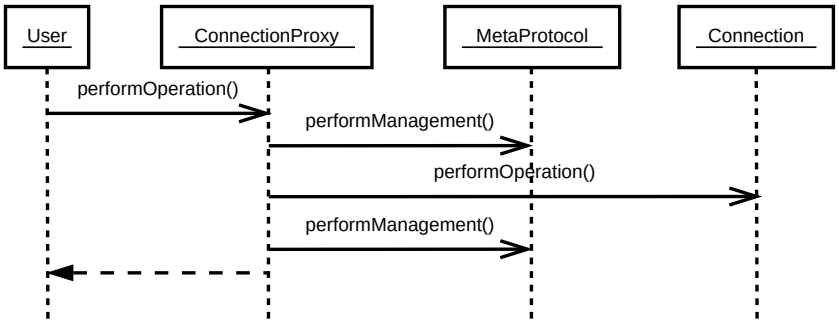

Figure 1.6: Metaprotocol.

This allows us to simulate the way some databases solve conflicts: granting locks to accessed objects (see figure 1.7). The general replication framework then should provide means to collect the readset and

writeset of the transaction at any time.

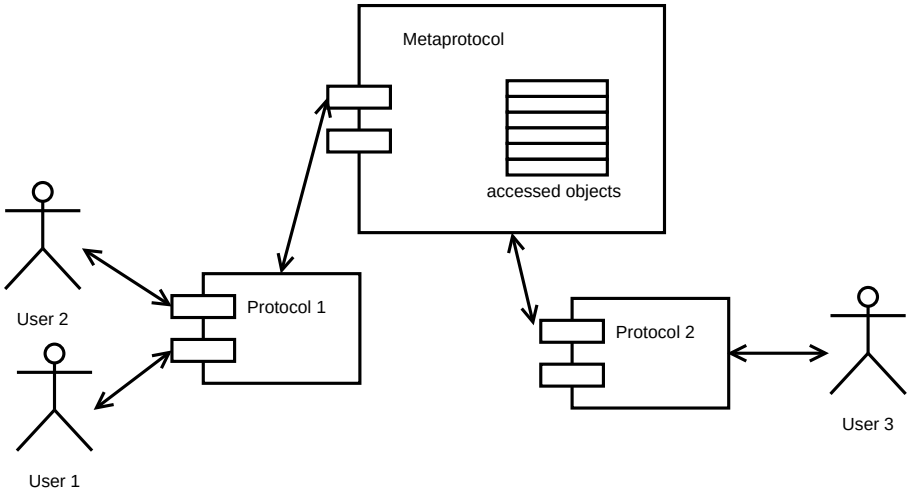

Figure 1.7: Common accesses space.

If the installed protocols and the underlying DBMS use the same isolation level local conflicts will be solved by the database itself. For multiple isolation levels handling an approach can be found at [11].
In any case, local operations may block remotely committed transactions and a mechanism to abort transactions even without the user interaction is needed.

Granularity

Granularity depends on the concept of object a protocol uses (column, group of columns, row, table, database). [7] uses rows as objects in its database model but also discusses about predicates and multiple isolation levels handling. As different protocols may use different object concepts an inclusion relationship between object classes is needed:

Definition: Object classes general inclusion relationship

Object class C_1 contains object class C_2 if all objects in C_2 are contained in C_1.

General inclusion is an internal operation between classes. Classes range from specific to general depending on how many classes they contain:

$$C_{\text{column}} \subset C_{\text{group of columns}} \subset C_{\text{table}} \subset C_{\text{database}}$$
$$C_{\text{row}} \subset C_{\text{table}} \subset C_{\text{database}}$$

However, we need an operation between objects and not classes to define all possible conflict cases. An internal operation is pretty straightforward:

Definition: Object conflict in the same class

o_a and o_b are two sets of objects of class C_i. o_a and o_b conflict $\iff o_a \cap o_b \neq \emptyset$.

Inclusion cannot be defined properly for classes between rows and columns: $C_{\text{group of columns}} \subset C_{\text{row}}$ cannot be established because columns select information “vertically“ (projection for all rows) and

rows select "horizontally" (select all for a single row). This means that saying that "*a row contains columns*" or that "*a column contains rows*" makes no sense in this context.
If fact, for objects, a row always intersects with a group of columns and vice-versa. This intersection defines the finest granularity in a relational database: a *row-column* when there is a single column, and then next finest group when there are several columns: a *row-column group*:

$$C_{\text{row-column}} \subset C_{\text{column}}$$
$$C_{\text{row-column group}} \subset C_{\text{group of columns}}$$
$$C_{\text{row-column}} \subset C_{\text{row}}$$

For all inclusion properties to hold the container table must be the same. Moreover, for the first one the column name must be the same, for the second one the column group must be the same, and for the third one the row must be the same.

An external inclusion operation can be defined listing all comparable cases but it is naturally derived if we define a class by a set of objects of a more specific class:

- A group of columns is a set of single columns. Each column contains the values stored in that column for all the table rows: A group of columns is the set of row-columns for all these rows.
- A row is the set of all row-columns for that row in its table.
- A table is the set of all its rows or the set of all its columns.
- A database is the set of all its tables.

Definition: Object conflict in different classes

o_a and o_b are two objects of class C_a and C_b. If $C_b \subset C_a$ then if objects in C_a can be defined as objects in C_b, internal object conflict operation result can be applied for this case. Otherwise, objects do not conflict.

1.3.5 Metadata maintenance

We classify metadata in two groups:

- **Basic**: These fields have to be collected for all protocols (even if they don't use metadata).
- **Additional**: These fields have a default value and they don't need to be maintained when the corresponding protocol is not being executed. As soon as it is, the default value is set and their management is started.

For the protocols that manage metadata information we need the MetaProtocol to maintain it. This maintenance is done similarly as the conflicts detection is.

Independently of the consistency protocol being used, all metadata maintenance has to be done. While collecting the writeset and readset, metadata for these objects has to be calculated. If this information can be obtained in an incremental way it will be easier to manage the information. If not operations over sets of objects would have to be performed extensively.

This is the main difference of metadata maintenance and conflicts detection and pure consistency logic. The latter ones are executed when necessary (when the metaprotocol is performing its tasks) while Metadata maintenance has to be always active:

Just before a transaction commits, it gets all metadata collected for all the consistency protocols and applies it.

1.3.6 Optimizations

A study has to be done to identify the stages where protocols need to call MetaProtocol. This information can be used to avoid certain calls from the proxy to the manager.

This approach is similar to AOP (Aspect Oriented Programming) paradigm. Here, protocol conflicts, metadata management and most consistency issues are considered as an aspect and they are implemented as a separate component, the MetaProtocol. An AOP framework could be used to implement these operations.

Chapter 2

Metaprotocol basic operations

We'll first show an algorithm based on [12] to stop protocols and the other operations (start and change) will be derived from it.

Our algorithms are placed in the MetaProtocol component. Once a message from the algorithm arrives to the Distributor, it redirects it to the Metaprotocol.

2.1 Stop protocol

The stop protocol uses a message called *stop*. This message is defined as follows:

$$\text{stop} = \{\text{list} = \text{vector}\, 1 \times N\}$$

where N is the number of configured nodes and each element represents each node stopping status.

Each node has a variable stopped that represents the different stopping stages a node goes through. Firstly, it is initialized to zeros for all alive nodes, this means that the protocol is not stopping. A 1 value means that the node has been notified about a stopping process. Finally, a 2 value means that the protocol is stopped in the given node. On the other hand, if stopped equals to 3 then the node is crashed[1].

The stopping state values are ordered in ascending priority. A stopping process will usually lead to a $0, 1, 2$ sequence for stopped values. When stopped equals to 3, no state changes have to be written for that node. Only when that node joins the system again, its stopping state values will change.
These state transitions can be modelled with an enumerate type:

```
enum StopStates {
  NOT_STOPPING,  // 0
  STOP_NOTIFIED, // 1
  STOPPED,       // 2
  CRASHED        // 3
}
```

If the stopping is initiated from node $localID$:

```
stopProtocol() {
  if (stopped[localID] == NOT_STOPPING) {
    stopped[localID] = STOP_NOTIFIED;
    bcast(stopped);
```

[1]The node is down or in a minority group.

```
    for all i in bcast_received do
      stopped[i] = STOP_NOTIFIED;
    done
  }
}
```

If the node is crashed (obviously in the minority group case), already stopping or stopped, there is no point initiating another stopping process.

If the message is received at another localID node:

```
receiveStop(stop st) {
  for all i in 1..N do
    stopped[i] = max(stopped[i], st.stopped[i]);
  done

  // 0 -> 1
  if (stopped[localID] == NOT_STOPPING)
    stopped[localID] = STOP_NOTIFIED;

  initiatorValue = st.stopped[st.stopInitiator];

  // when the st source starts the process or has just crashed
  // afterwards
  if (initiatorValue == STOP_NOTIFIED or CRASHED)
    order_to_stop();

  // when localID has just re-joined the system
  if (initiatorValue == STOPPED or CRASHED) {
    if for all i in 1..N stopped[i] == STOPPED or CRASHED
      remove_instance();
  }
}
```

receiveStop() orders the most common stopping state transition: from NOT_STOPPING to STOP_NOTIFIED ($0 \rightarrow 1$). In networkChanges() code, we can see that a node with a stopping state equal to STOPPED may receive more stop messages when a network reconfiguration is performed. In this case the STOPPED value will remain for that node.

As soon as order_to_stop() is called, user create transaction operations are blocked or aborted:

```
order_to_stop() {
  // Set as stopping and check whether the system is already stopped
  stopping = true;
  if (sessionCount == 0)
    notifier.notify();
}
```

remove_instance() sets stopping to *false* and performs all necessary operations to remove any runtime object from the working protocol from memory.

After a call to receiveStop() operation continues in the usual way. For each close() operation invoked over transactions MetaProtocol.closeSession() is called:

```
closeSession() {
  // Decrement the counter and check whether the system is already stopped
  stopping = true;
  sessionCount --;
  if ((stopping == true) && (sessionCount == 0))
    notifier.notify();
}

Thread Notifier {
  public void run() {
    while(true) {
      // Wait until there is a stopping process and the number of
      // current transactions reaches zero
      wait();
      if ((stopping == true) && (stopped[i] != STOPPED)) {
        stopped[i] = STOPPED;
        bcast(stopped);
    }
  }
}
```

This thread waits until all transactions in this node have been closed. It can be awakened when a termination condition becomes true. Both order_to_stop() and closeSession() contain termination condition verifications.

2.1.1 Fault tolerance and additional considerations

This protocol uses a communications primitive with very relaxed delivery guarantees. bcast() is assumed to be *unreliable*. An unreliable broadcast is equivalent to a sequence of unicasts. The proper way to implement this is as follows:

```
int[] bcast(Message m) {
  int[] nodes = getAliveNodesInCurrentView()
  Vector received = new Vector();

  for(int i=0; i<nodes.length; i++) {
    try {
      // send m point-to-point from localID to i
      send(m, i);
      received.add(i);
    } catch(NetworkException e) {}
  }

  return received.toArray();
}
```

A multicast method can easily be derived from the broadcast one. Multicast signature has the following aspect:

```
int[] bcast(int[] destinations, Message m);
```

Network changes might occur in the middle of a bcast() operation. Then the Membership component will send a notification to one of the MetaProtocol threads:

```
void networkChanges(int[] newNodes, int[] crashedNodes) {
  for i in crashedNodes do
    stopped[i] = CRASHED
  done

  // when localID is re-joining the system
  if (localID in newNodes) {
    stopping = false;
    stopped[locaID] = NOT_STOPPING;

  // when localID was already alive and during a stopping process
  } else if (stopping == true) {
    if for all i in 1..N stopped[i] == STOPPED or CRASHED
      remove_instance();

    // coordinator:
    else if (localID is the lowest previously alive node)
      bcast(stopped);
  }
```

Setting the stopped flag of crashed nodes to CRASHED for disconnections would not be necessary but this way, the algorithm is able to deal with node crashes[2].

After this, we need to check whether the algorithm has finished or not. If it hasn't, the "coordinator" sends a stopped order to the newly joined nodes.

Notice that most stopped element transitions triggered by networkChanges() broadcast will be from NOT_STOPPING to CRASHED. This happens due to the fact that nodes requested to stop don't know which nodes have received the stopping message and which have not.

It is not important as the only case where this would matter is when the stopping initiator crashes. When this happens, another "coordinator" broadcasts the stopping message. It has to broadcast and not multicast because some of the remaining alive nodes could have not been sent a stopping message yet.

In any case, we can provide the "coordinator" with a little bit more information. This is the only node that really knows which nodes have received the order and thus, it can write it down simply setting to STOP_NOTIFIED the stopped elements taken from the bcast() invocation:

```
  ...
    else if (localID is the lowest previously alive node) {
      bcast(stopped);

      for all i bcast_received do
        stopped[i] = STOP_NOTIFIED;
      done
    }
```

[2] In a node crash, the crashed node will have lost all stopping state information.

2.2 Start protocol

Starting a protocol is easier than stopping it. For the sequential case, once the previous protocol has been uninstantiated, a new protocol can be instantiated from any of the nodes. When the first message sent from this node to the others arrives, the protocol is used in the other nodes too as shown in figure 2.1.

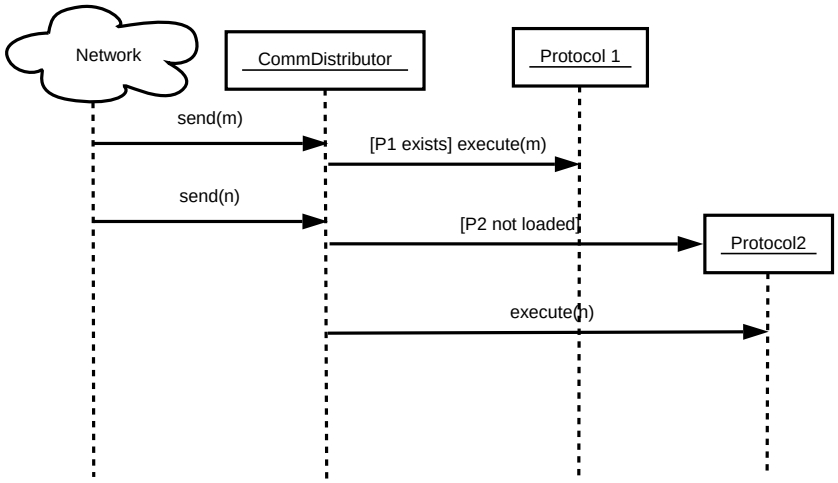

Figure 2.1: Start a new protocol.

When new transactions are blocked until the current protocol is stopped, they are blocked in their starting nodes until the nodes' stopped vector contains only STOPPED values.
Depending on the broadcast delivery order, some nodes may have perceived this situation before the last stop message has been delivered to the whole group. They will start operation and they might sent messages to nodes where the stopped vector completed with STOPPED values has not been established yet[3]. Incoming messages from a different protocol are blocked in those nodes too until the node is stopped in the same way as local operations are.

Considerations about dealing with long transactions when a protocol exchange is trying to be performed can be read in section 1.3.3.

When the protocol election can only be set from one of the nodes, this approach makes the use of an install_new_protocol message unnecessary.

A component to prevent protocol installation conflicts could be useful for some systems however, if we aim to a more general task, say allowing several protocols to co-exist, we will not try to prevent this situation and the start can be performed freely at any time.
The sequentiality control is performed when the start operation is invoked. Appendix A displays the algorithm that would be required for a sequential protocol exchange.

2.3 Change protocol

Changing a protocol can be done with a protocol similar to the *Stop protocol*: A change message has to be sent. This message contains a list and a protocol identifier.

[3]Remember that the broadcast message does not provide ordering guarantees.

Once a node receives the message, it loads the new protocol and from that moment on, the user calls to the proxy to create new transactions link the user to the new protocol: the proxy redirects the calls to the proper protocol.

At this point, both protocols coexist. As the goal is to change the protocol, the previous one must be stopped. The system needs to know which one is being loaded and which one is stopping. This can be achieved adding a protocol identifier to the stop message.

2.3.1 Other considerations

It is worth recalling that due to protocols basic metadata maintenance, the pieces of code in charge of metadata for all changeable protocols have to be running all the time. That is why protocol exchange leads to a concurrent multi-protocol environment.

To promote concurrency further when several protocols are running, communication and membership delivery calls have to be non blocking.

The exchange protocol simply lets transactions from the substituted protocol decrease because no characteristics from the involved protocols have been used at all.

Locking protocols share behaviour in pre-commit operations and we can take advantage of these similarities to optimize the protocol change. As soon as a *Change Point* has been reached, the proxy could be able to redirect a user call from one protocol to the other transparently and avoiding possibly long waiting intervals of time until the user decides to finish the transaction.

Previous to the metaprotocol design, an analysis of a set of protocols will be presented in the following chapter.

Chapter 3

Protocols metadata description

3.1 Protocols classification and duration descriptors

For the protocols classification we will use the models presented in [25] and [26]. [25] will be used to describe a protocol:

Architecture: Update everywhere / Primary copy.

Server interaction: Linear / Constant.

Transaction Termination: Voting / Non-voting.

In [26], authors offer a classification from the point of view of the protocol phases. We will use these phases to describe the different steps a transaction usually goes through:

Request (**RE**): The client submits an operation.

Server coordinator (**SC**): The replica servers (nodes) synchronize the operations execution.

Execution (**EX**): The operation is executed.

Agreement coordinator (**AC**): The nodes make an agreement about the execution result.

Response (**END**): The execution result is sent back to the user.

Another feature used to characterize a protocol is the transactional isolation level it is able to guarantee. As [11] has already referenced isolation level guarantees are based on phenomena definitions consisting of several object dependencies.

Moreover, when given we will use a classification according to the protocols metadata:

- **Object meaning or granularity:** Is the object a table row or the table itself?
- **Metadata context:** Object, transaction or group of transactions.
- **Metadata durability:** Is the metadata needed only during the life of the transaction or is it needed during the life of the object? Where, in the whole system, is really needed to be made persistent?

As we want this information to lead us to a general metaprotocol architecture we will pay attention to these fields of metadata with special detail. The following information for each metadata is important:

Name: Metadata identifier.

Description: Short metadata description when the name is not enough.

Duration: The context of the metadata: object, transaction and view. Where an "object context" attribute means that the metadata is valid during the life of an object and so on. Duration context identifiers will be included as basic types (OID, ViewID and TID) for our system and will be common for all protocols:

Object (oid):

$$\text{OID} = \text{repository} :: \text{class} :: \text{owner.counter}$$

Where:

int owner;	A fixed number of digits is used in order to separate owner from counter (values are left-padded with zeros).
long counter;	A sequentially incremented number.
String repository;	Database name with variable length.
String class;	Table name with variable length.

View (view_id): Integer value incremented each time there is a network reconfiguration.

Transaction (tid):

$$\text{TID} = \text{repository} :: \text{owner.counter}$$

Where fields have the same meanings and types than the previous identifiers.

Collection: Collected progressively (and during which phases) or in a single step.

3.2 Protocols metadata

3.2.1 SiDi protocols

FOB: Full Object Broadcast

FOB [23] was created and developed by the SiDi group at the ITI of the Politechnical University of Valencia (UPV) during the development of the *GlobData* project [2].

FOB is a *Primary copy*[1] – *Constant* – *Voting* protocol.

RE and EX occur in the primary copies between the begin transaction and the commit user calls. After the commit call, SC is acquainted through AC: Objects in the *wset* are organized into groups according to their node owner and a lock request is sent to each owner. If all owners answer affirmatively, an update is broadcasted using a reliable service. This message makes the primary copy to effectively commit the operations and return the control to the user (END). It makes the other copies to execute the transaction operations (EX) and to effectively commit too. Locks are released just after the effective commit.

FOB was implemented to guarantee several isolation levels provided by the middleware layer: PLAIN, CHECKOUT and TRANSACTION. Those being similar to *read committed*, *repeatable read* and *serializable*.

[1] Both FOB and COLU closest architecture definition is *Primary copy* but they are not exactly *Primary copy*: FOB and COLU can start transactions in any node and updates are first done in this starting node.

An object in FOB can be either a row or a table but we'll focus on the first case: a row. Metadata is collected progressively before commit and it is associated with a row whose primary key is an Object Identifier (oid).

Table 3.1 shows FOB metadata.

Name	Duration	Collection	Description
owner	Object	Progressive before commit and implicit in the oid	Original owner
version	Object	When needed	Object version
access_type	Transaction	Progressive before commit	Read / Write
lock_node	Transaction	During commit	Actual lock node holder
apply_into	Transaction	During commit	Nodes where it will be applied
object_info + SQL	Transaction	When effective commit	Apply info

Table 3.1: FOB metadata

Where:

int owner;	It can be derived from oid.
int version;	An integer value incremented each time a transaction updates and commits the object.
int lock_node;	During node crashes it may be different from owner.
int[] apply_into;	The list of nodes where the commit was broadcast. It is the whole list of nodes until there exist node failures.

COLU: Cautious Object Lazy Update

COLU [16] and [15] was created and developed by the same team of FOB. It is similar to FOB, it is *Primary copy – Constant – Voting* too, but it has several big differences:

- It multicast changes only to a subset of the nodes.
- During RE, an adaptive function is used to predict whether objects are outdated or not. When an outdated object is accessed, it is requested to its owner and it is subsequently updated before it's returned to the user.
- The same procedure is followed when, during AC, it decides a session must abort. This, and the previous one, are the main features that make the protocol a lazy one: Objects are mainly updated only when needed.

Table 3.2 shows COLU metadata.

Where:

double threshold;	The result of the adaptive function.
TStamp timestamp;	Instant when the last write commit took place.

3.2.2 *Dragon* protocols

The ETH group, within the context of the *Dragon* project, presented a suite of replication protocols based on group communication primitives.

The main idea of these protocols [20], [21] is to perform transactions locally and deferring writes to remote nodes until commit time. At this time, updates are broadcast using total order to guarantee the reception order. This way, no 2 Phase-Commit is needed.

Name	Duration	Collection	Description
owner	Object	Progressive before commit and implicit in the oid	Original owner
version	Object	When needed	Object version
access_type	Transaction	Progressive before commit	Read / Write
threshold	Transaction	Before commit	Prediction value per remote object
timestamp	Object	When write accessed	Last write commit
lock_node	Transaction	During commit	Actual lock node holder
apply_into	Transaction	During commit	Nodes where it will be applied
object_info + SQL	Transaction	When effective commit	Apply info

Table 3.2: COLU metadata

SER: Replication with Serializability

SER is an *Update everywhere – Constant – Non-Voting* protocol. It provides *1-copy-serializability* consistency.

During RE, it acquires local read locks while write requests are deferred until the beginning of the SC phase. In SC, writes are sent using a total order multicast and AC begins when locks are being granted. EX happens just after locks are granted and END happens when the effective commit or rollback is performed.

Authors mention that a transaction reads or writes logical objects. It can be derived from the described isolation levels that they work with table rows and units of information. Object identifiers are progressively collected before commit.

Table 3.3 shows SER metadata.

Name	Duration	Collection	Description
access_type	Transaction	Progressive before commit	Read / Write
apply_into	Transaction	During commit	Nodes where it will be applied
object_info	Transaction	When locks are granted	Apply info

Table 3.3: SER metadata

Where:

`int[] apply_into;` The list of nodes where the commit was broadcast.

CS: Cursor Stability

Cursor stability introduces the notion of *short read locks* to avoid transaction starvation. It is extended straight from SER using a third kind of access lock.

CS is also a *Update everywhere – Constant – Non-Voting* algorithm. It does not provide serializability as dirty reads, lost updates and write skews might occur.

Table 3.4 shows CS metadata.

Name	Duration	Collection	Description
access_type	Transaction	Progressive before commit	Read / Short Read / Write
apply_into	Transaction	During commit	Nodes where it will be applied
object_info	Transaction	When locks are granted	Apply info

Table 3.4: CS metadata

SI: Snapshot Isolation

SI was defined to avoid read locks. It is very similar to SER and it can be described using the same terms used for SER and CS but it uses timestamps and object versions to allow database multiversioning. As described by its name, it provides snapshot isolation guarantees.

Table 3.5 shows SI metadata.

Name	Duration	Collection	Description
BOT	Transaction	Immediately when stated	Begin Of Transaction
access_type	Transaction	Progressive before commit	Read / Write
version	Object	When granting lock	Last transaction that wrote the object
apply_into	Transaction	During commit	Nodes where it will be applied
object_info	Transaction	When locks are granted	Apply info
EOT	Object	Immediately when committed	End Of Transaction

Table 3.5: SI metadata

Where:

TStamp BOT;	Logical instant when the transaction started.
int version;	An integer value incremented each time a transaction updates and commits the object.
int[] apply_into;	The list of nodes where the commit was broadcast.
TStamp EOT;	Logical instant when the transaction finished.

3.2.3 Lin – Kemme and Patiño – Jimenez

SI-Rep

SI-Rep [22] is a more detailed work about snapshot isolation protocols whose origin is [21]. It contemplates several implementation issues such as an ws_list for conflict resolution and a tocommit_queue to apply transactions.

Table 3.6 shows SI-Rep metadata.

Where:

int version;	An integer value incremented each time a transaction updates and commits the object.
int[] apply_into;	The list of nodes where the commit was broadcast.

3.2.4 El Abbadi – Toueg

[6] presents a replica control protocol on top of any concurrency control protocol for conflict detection. The protocol provides *1-copy-serializability* using view ordering. Conditions for object access and network

Name	Duration	Collection	Description
BOT	Transaction	Immediately when stated	Begin Of Transaction
access_type	Transaction	Progressive before commit	Read / Write
version	Object	When granting lock	Last transaction that wrote the object
apply_into	Transaction	During commit	Nodes where it will be applied
object_info	Transaction	When locks are granted	Apply info
EOT	Object	Immediately when committed	End Of Transaction

Table 3.6: SI-Rep metadata

partition situations are explained using accessibility thresholds ($A_{r/w}$) an quorums ($q_{r/w}$) where:

$A_{r/w}$: Minimum number of copies available in a view in order to access an object.

$q_{r/w}$: Minimum number of copies to physically access in order to write or read an object in a view.

Table 3.7 shows El Abbadi–Toueg metadata.

Name	Duration	Collection	Description
access_type	Transaction	Progressive for *rset*	Read / Write
sites	Object	Predefined	Nodes where the object exists
$A_{r/w}$	View	When accessed	Accessibility thresholds
$q_{r/w}$	View	When accessed	Quorum (physical accesses needed)
version	Object	When granting lock	(view, sequence)

Table 3.7: El Abbadi–Toueg metadata

Where:

int[] sites;	It is a predefined attribute.
IntTuple Arw;	(Ar, Aw): number of read/write existing copies for a given object in the current view.
IntTuple Qrw;	(Qr, Qw): physical accesses needed to access a given object in the current view.
IntTuple version;	Object oid has been written sequence times during view.

For this protocol we define a basic type called IntTuple containing two integer values: (int, int).

We also modify a little bit the definition that authors give of version. In [6], if sequence is k and view is v_id then it means that if t was the last transaction to write the object it is the kth transaction to write it during v_id. As we allow our architecture to encapsulate several sequential transactions inside the scope of the same Transaction object (representing in this case what we called a session), we rewrite the definition as:

... it is the kth time a transaction has committed the object during v_id.

3.2.5 Agrawal – El Abbadi – Steinke

A family of epidemic algorithms based on the causal delivery of log records is given in [8]. These are naive, pessimistic and optimistic.

Naive protocol

This family of protocols is epidemic due to the nature of the write changes communication procedure. The naive protocol guarantees *1-copy-serializability* by avoiding concurrent transactions (delaying them when possible or aborting them otherwise).

During RE, the transaction is executed (EX) on a single node and acquires object locks. The timestamp is picked up when SC starts in a way that causality of transactions is guaranteed. During SC operations are sent in an epidemic process that might lead to the abortion or commitment of the transaction. This means that the protocol is *Voting*. It is also *Update everywhere* and *Linear* because the epidemic process can be bounded.

Table 3.8 shows Naive Agrawal–El Abbadi–Steinke metadata.

Name	Duration	Collection	Description
timestamp $TS(t)$	Transaction	When started	$T[i, *]$
access_type	Transaction	Progressive before commit rset released when commit	Read / Write
wset_values	Transaction	Progressive	*wset* apply info
site	Transaction	When started	Node where the transaction started
abort / commit flag	Transaction	When result is decided	Result

Table 3.8: Naive Agrawal–El Abbadi–Steinke metadata

Where:

TStamp tst;	Beginning of transaction.
Object_Info wset_values;	Equivalent to object_info.
int site;	It can be derived from tid.
boolean abort;	Established when the user commit request is resolved.

[8] declares $T_i[k, j] = v$ as follows:

> Node i knows that node k has received the records of all events at node j up to time v.

Timestamp calculation for this protocol is a little bit more complicated than in the rest of studied protocols. Authors explain in the paper how to propagate this information from node to node: it is sent when node i sends a message to node k.

Pessimistic protocol

The pessimistic protocol derives information from T_i in order to avoid the abort / commit flag.

Table 3.9 shows Pessimistic Agrawal–El Abbadi–Steinke metadata.

Optimistic protocol

The optimistic version of the family of epidemic algorithms is managed via an optimistic releasing of locks. It is designed so that it still guarantees serializability.

Name	Duration	Collection	Description
timestamp $TS(t)$	Transaction	When started	$T[i, *]$
access_type	Transaction	Progressive before commit rset released when commit	Read / Write
wset_values	Transaction	Progressive	*wset* apply info
site	Transaction	When started	Node where the transaction started

Table 3.9: Pessimistic Agrawal–El Abbadi–Steinke metadata

Name	Duration	Collection	Description
timestamp $TS(t)$	Transaction	When started	$T[i, *]$
access_type	Transaction	Progressive before commit rset released when commit	Read / Write
wset_values	Transaction	Progressive	*wset* apply info
site	Transaction	When started	Node where the transaction started
inconflict flag	Transaction	When result is decided	Conflict resolution request
readfrom	Transaction	During read compilation	Transactions from which the transaction reads from

Table 3.10: Optimistic Agrawal–El Abbadi–Steinke metadata

Table 3.10 shows Optimistic Agrawal–El Abbadi–Steinke metadata.
Where:

boolean inconflict;	similar meaning that abort from 3.2.5.
TID[] readfrom;	Active transactions that have previously accessed objects accessed by the current one.

3.2.6 Jimenez – Patiño – Kemme – Alonso

In [18] a protocol based on optimistic delivery broadcast primitives[2] that guarantees *1-copy-serializability*. Queries (*rset*) are executed only at the local node using snapshot isolation.

The protocol is able to distinguish between what authors call classes. A class can be either a tuple or a selection over a table.

When the transaction starts, it is broadcast to all nodes but only the starting node executes it. RE happens before this and later, SC starts and EX too (only in the starting node). After this execution, a commit message including the update information is sent to all the nodes so that EX happens everywhere. AC is performed only at the starting node.

Table 3.11 shows Jimenez–Patiño–Kemme–Alonso metadata.
Where:

boolean executed;	Operations finished.
boolean commitable;	to-deliver sent.
SQL[] update_info;	Transaction update sentences.

[2] to-broadcast(m), opt-deliver(m) and to-deliver(m).

Name	Duration	Collection	Description
executed	Transaction	When the execution finishes	Finished execution
commitable	Transaction	Transaction to-delivered	Transaction to-delivered
access_type	Transaction	When started	Read / Write
update_info	Transaction		*wset* apply info

Table 3.11: Jimenez – Patiño – Kemme – Alonso metadata

3.2.7 Pacitti – Minet – Simon

The protocols described in this section (deferred and immediate) were presented in [24]. The main difference between both is the updates propagation.

Both protocol versions are based on the existence of a global FIFO reliable multicast with a known upper bound given by ϵ, a constant that limits the nodes' clocks synchronization.

Deferred protocol

When deferred, updates are propagated after commitment with a single message. Then, this version is *Constant*.

Table 3.12 shows Deferred Pacitti–Minet–Simon metadata.

Name	Duration	Collection	Description
timestamp	Transaction	When committed	Commitment time
update_info	Transaction		Write operations

Table 3.12: Deferred Pacitti – Minet – Simon metadata

Where:

TStamp timestamp; Commitment time.
SQL[] update_info; Update SQL sentences.

Immediate protocol

When immediate, each operation is multicast before commitment. The protocol is *Linear*.

Table 3.13 shows ImmediatePacitti–Minet–Simon metadata.

Name	Duration	Collection	Description
timestamp	Transaction	When committed	Commitment time
update_info	Transaction	When executed	Write operations

Table 3.13: Immediate Pacitti – Minet – Simon metadata

3.2.8 Summary

Table 3.15 shows a map of the protocols and associated metadata for a first approach on finding common metadata. Columns represent different protocols, whose acronyms can be found in table 3.14, metadata names and metadata context or duration.

Acronym	Name
SiDi	
FOB	Full Object Broadcast
COLU	Cautious Lazy Update
Dragon	
SER	Serializability
CS	Cursor Stability
SI	Snapshot Isolation
SI-R	Lin–Kemme–Patiño–Jimenez
Abb	El Abbadi–Toueg
Naive	Naive Agrawal – El Abbadi – Steinke
Pes	Pessimistic Agrawal – El Abbadi – Steinke
Opt	Optimistic Agrawal – El Abbadi – Steinke
Jim	Jimenez – Patiño – Kemme – Alonso
Def	Deferred Pacitti – Minet – Simon
Imm	Immediate Pacitti – Minet – Simon

Table 3.14: Acronyms for protocols

Table 3.15 has been compiled using information from tables 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12 and 3.13. When a metadata field is required by a protocol, this fact is identified by a "X" symbol. When the "X" symbol is replaced by a different identifier, say *id*, this means that the field represented by this row and *id* are equivalent or that it can be derived from *id*.

3.2.9 Basic metadata

Basic metadata was introduced in 1.3.5 and it is composed of the minimum set of metadata attributes that could be considered common. This means that whatever protocol is using the system, basic metadata is updated for all the objects.

Logically, the more protocols we study the more difficult finding common properties will be. This fact limits an ideal basic metadata set to:

$$\{\text{oid}, \text{access\_type}, \text{tid}, \text{update\_info}\}$$

Being update_info an object from a class that encapsulates the tid SQL sentences as well as operations that allow to update its result set objects independently.

Notice that for a protocol to start working, its metadata has to be ready. If it is not, and no default value can be set, then it must be calculated. Forgetting about additional metadata of the protocols that are not currently during execution and performing this calculation at their start time would have two big disadvantages:

- Latency: *All* the database objects *additional metadata* has to be updated previous to the new protocol installation.
- Artificial and mistaken values: It may be impossible to establish metadata values if it is not during the precise moment when they are required.

Metadata	FOB	COLU	SER	CS	SI	SI-R	Abb	Naive	Pes	Opt	Jim	Def	Imm	**Duration**
tid	X	X	X	X	X	X	X	X	X	X	X	X	X	*Transaction*
oid	X	X	X	X	X	X	X	X	X	X	X	X	X	*Object*
owner	oid	oid												*Object*
version	X	X			X	X	X							*Object*
access_type	X	X	X	X	X	X	X	X	X	X	X			*Transaction*
lock_node	X	X												*Transaction*
apply_into	X	X	X	X	X	X								*Transaction*
threshold		X												*Transaction*
timestamp		X												*Object*
object_info	X	X	X	X	X	X		wset_val	wset_val	wset_val	upd_nfo	upd_nfo	upd_nfo	*Transaction*
BOT					X	X		$TS(t)$	$TS(t)$	$TS(t)$				*Transaction*
sites							X							*Object*
$A_{r/w}$							X							*View*
$q_{r/w}$							X							*View*
abort/commit								X		inconflict				*Transaction*
site								tid	tid	tid				*Transaction*
readfrom										X				*Transaction*
executed											X			*Transaction*
commitable											X			*Transaction*
EOT					X	X						boc	boc	*Transaction*

Table 3.15: Metadata summary

The second disadvantage is better explained with an example:

> Consider that we are stopping the current protocol, say P_0, and we want to use **COLU**. If timestamp is not set during the execution of P_0, at the changing time all the database metadata has to be updated and then timestamp can be set as an ancient date or as a recent one. In both cases we will lose timestamp real meaning for a while until **COLU** is being executed for a long time and it makes sense. Not until then, all predictions calculated by **COLU** using timestamp would be accurate again.

This explains why, when not default value exists, the maintenance of additional metadata is necessary for the metaprotocol to be correct.

Chapter 4

Architecture

All protocols are meant to be implemented under a scenario similar to the one described in section 1.2. A middleware layer or a component embedded inside the database manager is used for this purpose. Regardless of global performance issues, protocol designers pursue further considerations such as transparency and low overhead.

Low overhead is tightly related to metadata. Metadata is always meant to be easily obtained and to minimize the required storage space. Transparency means that user API for data retrieval has to be kept untouched or minimally changed.

Independently of the metaprotocol option chosen for protocols concurrency, in order to provide a fast change, metadata has to be available whenever the protocol change is requested. This suggests that all metadata for all protocols has to be maintained even when the protocols are not loaded. Common attributes are naturally maintained during transactions execution and particular ones will be worked out as if it were a background process.
We choose an approach like this one, where metadata is kept updated, because it allows finer *Change Points* granularity. If an object is accessed by two consecutive transactions, each one using different protocols, the second one will have accurate metadata values (for example, version or timestamp). Thus it guarantees that changing a transaction protocol into a different one can be done.

4.1 Metadata structures

In a data access API such as Java's *JDBC*, a Transaction executes queries and updates and obtains several ResultSet objects. ResultSets contain the accessed objects and ResultSetMetaData can be used to obtain table column names and types.

Based on table 3.15, a similar structure for the protocols metadata will be used.

A transaction will be encapsulated inside a Transaction object. We will not take into account that several queries and updates return different ResultSets, we will consider a single ResultSet per transaction that contains the union of all executed operations' ResultSets. Values (actual data) for the accessed objects are only necessary for the user, the protocols work strictly with metadata. However metadata needs to keep track of this information in order to broadcast it where necessary.

In the end, we will have two structures in charge of all this information ObjectMetaData and TransactionMetaData. Both implement a general MetaData interface which, at least, allows information to be

serialized.

As most protocols allow the incremental construction of the ResultSets, MetaData will be able to be constructed incrementally, too.

```
class TransactionMetaData implements MetaData {
  String tid;             // repository::owner.seq_id
  ObjectMetaData[] rset;  // Readset               = access_type
  ObjectMetaData[] wset;  // Writeset              = access_type
  ObjectMetaData[] srset; // Short ReadSet for cs  = access_type
  TStamp bot;             // si, agr protocols (TS)
  TStamp eot;             // si, pac protocols (timestamp)
  TStamp boc;             // def-pac, imm-pac (EOT)
  SQL[] execution;        // User's SQL sentences  = object_info
  String[] readfrom;      // opt-agr
  boolean executed;       // jim
  boolean commitable;     // jim
  boolean abort;          // naive-agr and opt-agr (inconflict)

  public String getRepository(); // derived from tid
  public int getNodeOwner();     // derived from tid
                                 // site for agr protocols is obtained from owner
}
```

This structure for Transactions has an identifier, *tid*, that acts as a primary key.

access_type can be obtained from the lists: rset, srset and wset.

ObjectMetaData lists and execution build what in table 3.15 was called object_info. They belong to the basic metadata and are compulsory for the TransactionMetaData construction. However, the ObjectMetaData lists will be built gradually and so ObjectInfo for these lists will.

getNodeOwner() is the way to obtain the site value for Agrawal et al. It is an option that all protocols have to obtain the node where the transaction was started.

Timestamps are typically used for ordering purposes and the TStamp data type has to provide means to maintain this order in a protocol independent manner. TStamp may hold a vector clock that is able to store [8]'s attribute. For total replication protocols there exist cheaper solutions such as a counter of applied transactions.

```
class ObjectMetaData implements MetaData {
  String oid;             // repository::table::owner.seq_id
  int version;            // fob, colu, si, si-rep, abb
  int lock_node;          // fob, colu
  int[] apply_into;       // For recovery issues in most protocols
                          // fob, colu, ser, cs, si, si-rep
  ObjectInfo info;        // Information to apply a single object
                          // wset_val for agr protocols
  long timestamp;         // colu
  int[] sites;            // abb
  double threshold;       // colu
  IntTuple Qrw;           // abb
```

```
    IntTuple Arw;           // abb

    public int getNodeOwner();     // derived from oid
    public String getRepository(); // derived from oid
    public String getTable();      // derived from oid
  }
```

For ObjectMetaData only the identifier, *oid*, is a common field for all the studied protocols.

getNodeOwner() is the way to obtain the owner value for SiDi protocols. info for wset objects contains information for Agrawal wset_value, for Jimenez et al. and Pacitti et al. update_info and for all protocols requiring update information.

timestamp holds **COLU**'s timestamp attribute. This value is used independently in each node to obtain access prediction values from a given formula based on real time differences. That's why it does not use TStamp data type because it only allows logical order and has not enough information to deal with time measures.

In both structures care must be taken for the data types of metadata fields. For info an effort should be done in the middleware architecture in charge of protocol management to provide a same data type for objects serialization regardless of the protocol.
Another case is El Abbadi–Toueg's version field which holds a tuple (view, sequence) while for the other protocols needing version information this field holds integer values. In these cases there are two options: the best one is to find a bidirectional conversion between both data types. If it was not possible then the fields should not be grouped.
Yet another situation is $TS(t)$ for Agrawal–El Abbadi–Steinke protocols. This field acts as a timestamp and has different representation for BOT than **SI**. In this case it is convenient to consider the rest of timestamps too (EOT, BOC) and try to find a common representation, such as vector clocks or logic counters, in order to make them comparable.

4.1.1 Completion and serialization

Serialization of the metadata structures depends on the context of each metadata attribute. The only not null value of each structure is *oid* and *tid*. As soon an object is created, the associated *ObjectMetaData* must be created.

For a sequential change of protocols completion is only needed for metadata available during object context. The rest of information is useful for a protocol change in between an active transaction and, for the sequential case, it can be set to default values.

4.2 Architecture

Figure 4.1 shows the classes model of the metaprotocol architecture. Some components depicted in previous sections have been deployed into several parts. For example, figure 1.4's *MetaProtocol* is now behind the *ProtocolProxy* and accesses a set of *MetaDataManagers*, one per protocol. In fact, the *MetaProtocol* concept exposed in the previous sections represents all these new components together.

First of all, we explain each component separately:

Transaction: It is the object a user obtains to access data.

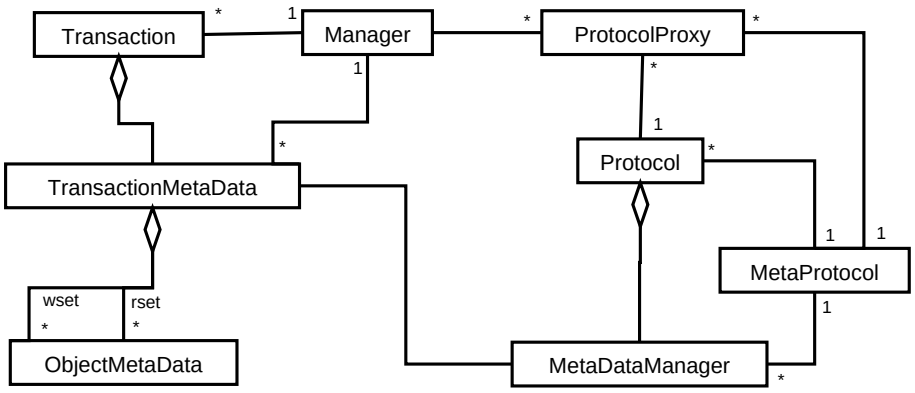

Figure 4.1: Architecture classes model.

Manager: The node's manager component is the core of the replicated system. It isolates the user from any replication detail.

TransactionMetadata: See section 4.1.

ObjectMetaData: See section 4.1.

ProtocolProxy: It acts as a *Distributor* (see figures 2.1 and 1.3).

Protocol: It contains the consistency protocol logic.

Metaprotocol: It contains protocol coordination logic.

MetaDataManager: It contains the metadata management logic of the consistency protocols.

The *Manager* creates transactions and attaches each of them to their corresponding metadata objects for transactions and objects (*TransactionMetadata* and *ObjectMetadata*). The main task of the *Manager* is to coordinate the work requested by the *Transactions* and to let the *Protocols* take care of consistency. The protocols are "hidden" behind their corresponding *ProtocolProxy* objects (see section 4.2.1).
When invoked, the *MetaProtocol* will take care of the protocols coordination. One of its tasks, metadata maintenance, is performed by the *MetaDataManager* objects.

4.2.1 Protocol encapsulation

Figure 4.2 shows the part of figure 4.1 that explains how a protocol is split into.

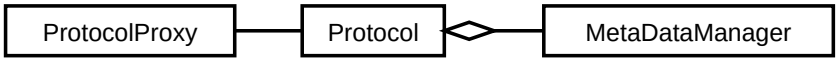

Figure 4.2: Protocol components.

The *ProtocolProxy* decides which *Protocol* executes the operation. Each *Protocol* has a *MetaDataManager*. Isolating metadata manager details in a separated component allows to keep all protocols metadata managers running.

Observing the collection column of the protocols metadata tables in section 3.2 it can be seen when the protocol metadata managers have to be invoked. This invocation must be done through the *MetaProtocol* component because this piece of the architecture is in charge of the administrative events coordination.

Both *ProtocolProxy* and *Protocol* can invoke *MetaDataManager* operations. However code will be clearer if one of the two components is selected for this function.

Placing calls in *ProtocolProxy* emulates the AOP paradigm of "@before" and "@after" invocation clauses. For example:

```
class COLUProxy implements ProtocolProxy {
  ...
  Protocol p;
  ...
  protected void changeProtocol(Protocol newp) { p = newp; }

  public boolean updateObjects(Object[] writes) {

    // @before operations
    metaproto.setTimestamp(this, writes);

    // actual operation
    boolean result = p.updateObjects(writes);

    // @after operations
    metaproto.getVersions(this, writes);

    return result;
  }
}
```

However, this might be not enough in all cases and a "@meanwhile" clause may be needed. We can obtain this placing the metaprotocol calls inside the *Protocol*.

The invocations contain the *ProtocolProxy* itself for the metaprotocol to know the source of the method call and to optimize *MetaDataManager* accesses.

The metaprotocol is the core of the main study presented here. It will hold the management algorithms (see chapter 2) and coordination logic. This coordination logic includes structures to perform conflict access resolution operations.

4.2.2 Concurrency options

Figure 4.3 shows an object diagram for a case with a user and two open transactions. Here, it can be seen that the *Manager* and the *MetaProtocol* are common and unique.

It can also be seen that all *MetaDataManagers* are always loaded and fill the contents of the transaction and objects metadata.

Figure 4.3 shows that the *Protocol* component is also common but this situation is induced by the fact that the figure depicts a sequential protocol changing scenario.

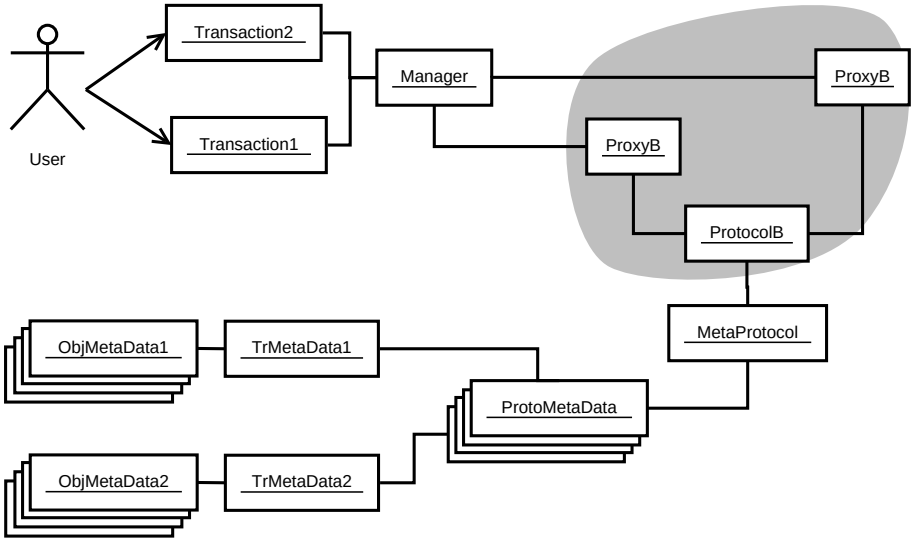

Figure 4.3: Architecture for sequential changes object model.

Allowing the architecture to be able to hold several protocols concurrently is obtained by having the proxies to be able to decide independently from each other about the encapsulated *Protocol* they invoke (see figure 4.4).

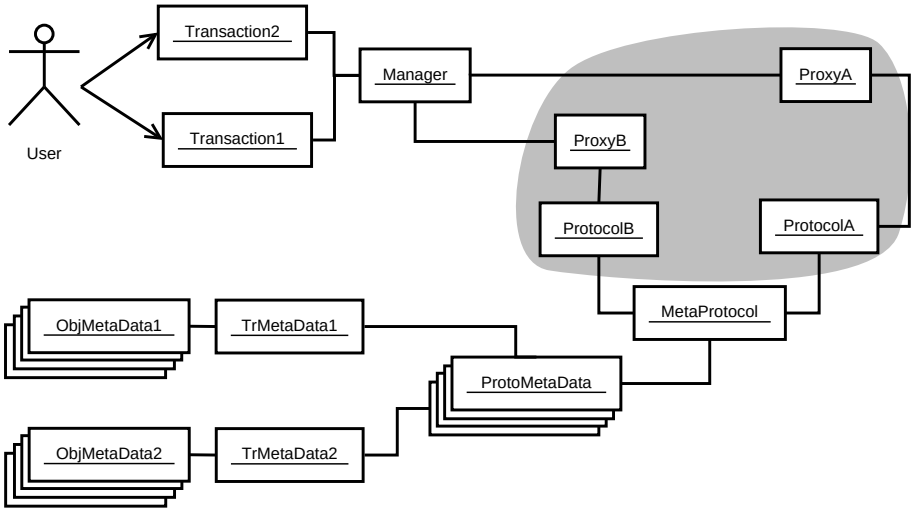

Figure 4.4: Architecture for parallel changes object model.

4.2.3 Implementation considerations

While some authors explain about commit queues, others skip this information due to its low level nature. When the network message reception frequency surpasses the node's capacity to apply updates and perform commit operations, commit queues hold the to-be-committed messages that are waiting in the destination node.

To enforce concurrency most protocols consider that a message is applied as soon as it is received but this is not always true specially when hardware characteristics differ, load is not balanced or due network

topology.

We consider this fact out of the scope of the metaprotocol.

4.3 Conflicts detection and resolution

Recalling the concepts outlined in section 1.3.4 conflicts detection is based on the comparison of transaction Result Sets. For conflicts resolution, the following methods are called extensively:

```
public void accessObjects(TransactionMetaData tmd, ResultSet rs)
  throws ConflictException;

public void releaseObjects(TransactionMetaData tmd, ResultSet rs);
```

These operations allow us to work over scenarios similar to the one depicted in figure 1.5 from page 9. Initially, all we need to know is contained in the methods signature: when the methods are invoked, for each object in rs tmd.tid is stored. The result sets are passed and stored incrementally. That is why rs is passed and it is not obtained from tmd itself, because it is a subset of $\{\text{tmd.rset}, \text{tmd.wset}, \text{tmd.srset}\}$.

It is pertinent to pass the TransactionMetaData tmd because we can reach the transaction from tmd. It could be possible that a transaction with a stronger isolation mode requested already granted accesses and the previous ones needed to be revoked. Using tmd, the transaction can be aborted at any moment and not only during the call to accessObjects.
For this means what we need first is an internal operation capable to abort a database transaction at any moment; such an operation exists in [4] middleware. Lastly, as the protocol is loaded, its abort process would be used to notify all the required nodes.

For further explanations assume we have the following variables:

```
ProtocolID id1, id2;
```

It is obvious that if $\text{id}_1 = \text{id}_2$ then access conflicts are resolved by the protocol itself. What is not obvious is the fact that for $\text{id}_1 \neq \text{id}_2$ the same applies because several additional actions need to be taken into consideration. These actions form the core of the metaprotocol. We will explain them with a simple example and after that a general case will be shown using pseudo-code notation.

Let's say we have transactions t_1 and t_2 with $t_1.protocolID() = \text{id}_1$, $t_2.protocolID() = \text{id}_2$ and $\text{id}_1 \neq \text{id}_2$. Imagine the following two user operations executed each one inside one of the transactions and executed possibly by different users in different nodes:

```
ResultSet r = t1.execute(SQL1);
...
ResultSet s = t2.execute(SQL2);
```

Internally, before the user call returns a value, t_1's last operation obtains a result set r and then it executes the following operation successfully:

```
accessObjects(t1.metadata, r);
```

Similarly, before `releaseObjects(t1.metadata, r)` is invoked, a t_2 operation obtains a result set s where $r \cap s \neq \emptyset$ and executes:

```
accessObjects(t2.metadata, s);
```

In case any of the protocols broadcasts the object access operation and accessObjects() for t_2 is executed in a node where accessObjects() for t_1 was previously executed, then id_2 decides what to do with t_2 (grant access, queue or deny and abort). After the invocation of these methods each object holds a list of transactions accessing it and the access mode (read, write and short read). These lists are called *object access queues*. For a given transaction isolation level none of the protocols overrides an already granted write access but a write can be queued behind previous read accesses. A write can be queued behind previous writes when there is a database lock before the actual commit.

At this point, if any of the transactions aborts, the conflict resolution problem between different protocols is done. If both transactions continue then any of them will try to commit effectively.

Imagine both t_1 and t_2 want to commit. As we include different delivery guarantees and lazy protocols, as soon as id_1 and id_2 decide that they have all the guarantees needed to commit, they send a *total order broadcast* to all nodes containing the following information: $\{\text{tid}, \text{tofinish} = \text{true}\}$.
This message is necessary and its delivery guarantees are necessary too:

> We cannot wait for the *effective commit* request message to broadcast this information because *we need to order the transactions execution in all nodes*.

The message implies an intention to commit. Nodes keep a list of *tofinish* transactions and accesses to an object granted for a *tofinish* transaction will never be revoked.

For this example, let t_2 be ordered before t_1. Depending on the transaction isolation mode (this management is done by the protocols themselves), as soon as the *tofinish* message is received in a node, transactions in its object access queues containing a t_2 entry are aborted (all of them except t_2). To improve the metaprotocol efficiency these abortions can be requested too as soon as the *tofinish* message is received (notice that this work would have been done afterwards if launched when the effective commit message is received).

After the *tofinish* total order broadcast, the effective commit process continues and the transactions have to face one of two possible situations:

A) There is no *tofinish* message from another transaction or it was ordered after t_2's one.

B) Another transactions' *tofinish* message was ordered before t_2's one.

For transaction t_2 the case is situation A) and the effective commit request is broadcast in id_2 protocol way so that t_2 finally commits. For transaction t_1 the case is situation B) and the transaction finally aborts. The t_1 commit request does not need to be sent. If it had already been sent it would be discarded upon reception in all the required nodes.

4.3.1 Inter-protocol conflict resolution protocol

In this section we will describe the example above for a general case using pseudo-code pieces:

```
public void accessObjects(TransactionMetaData tmd, ResultSet rs) {
  ...
  // For all objects in the partial result set
  for i in rs do
    set_access(rs[i], tmd.tid, rs[i].access_type);
  ...
}
```

Where set_access fills the object access queues.
releaseObjects() performs the opposite operation:

```
...
// For all objects passed in the parameter
for i in rs do
  set_access(rs[i], tmd.tid, none);
...
```

After the operations are done the user requests a commit:

```
public boolean commit_request() {
  ...
  boolean result = check_all_conflicts();

  if (!result) abort(tid);
  else {
    result = request_effective_commit(tid);
    if (result)
      result = transaction_result(tid);
  }

  return result;
}
```

check_all_conflicts() is protocol dependent and works with sets of objects as if they had been granted by the same protocol.

As it can be seen in the next bit of pseudo-code, effective_commit() tries to avoid the commit request delivery when it is already known that the result is to rollback the transaction:

```
public boolean request_effective_commit() {
  boolean result = false;
  ...
  send(total_order_bcast(tofinish(tid)));
  ...
  if (!tofinish_before(tid)) {
    bcast(effective_commit(tid));
    result = true;
  }

  return result;
}
```

Another thing that the commit_request() code shows is that the transaction final result is collected in the transaction_result() method. This method waits for the effective_commit message to be received and processed in the origin-of-commit node. This is necessary because other previous tofinish messages may have not been delivered yet.

tofinish_before return value condition is explained as:

Abort tid if there is another transaction whose result set interferes with tid's result set and this access was granted before tid's access was requested.
Otherwise allow tid to commit.

Obviously this condition will need to take into consideration transaction isolation properties and the existence of *short readsets*.

tofinish_before() is always a local call:

```
public boolean tofinish_before(String tid) {
   String[] grants;

   // For all objects in tid's complete result set
   for i in tid.rs do {
     // Get an ordered list of access grants
     grants = getTransactionAccessing(rs[i]);

     tid_pos = index of tid in grants or -1 if it does not appear;
     for j in grants where tofinish[j] == true do {
       other_pos = j transaction position in grants;
       if ((j <> tid) && (other_pos<tid_pos))
         return true;
     }
   }

   return false;
}
```

When the total_order_bcast(tofinish(tid)) message is received, the following code is executed:

```
public void tofinish_before_reception(String tid) {
   tofinish[tid] = true;

   if (tofinish_before(tid))
     abort(tid);
   else
     abort_conflicting_transactions(tid);
}
```

The abort(tid) call in this code does not need to send an abort communication message to any of the other nodes. The tofinish total order broadcast has already provided enough information everywhere for the nodes to work out the commit result independently.

The following code describes the steps taken when an effective_commit message is processed at any node:

```
public boolean effective_commit_reception() {
  ...
  boolean disard = tofinish_before(tid);
  if (!discard) {
```

```
      abort_conflicting_transactions(tid);
      execute_commit(tid);
    }

    return discard;
  }
```

transaction_result() waits for this method to finish and then it returns the real transaction commit or rollback state.

Now that the protocol is described it is worth considering some interesting facts:

- Notice that this code pieces are embedded inside the consistency protocols logic. All protocols process a commit_request user operation and use internal equivalents for the "effective_commit" methods. The metaprotocol methods simply wrap the original protocols methods.
- set_access() does not necessarily mean that we are using locks or that we actually execute the operations before commit (remember that some protocols delay this execution until the commit_request() call).
  set_access() aim is to populate the information required for latter intersection of sets queries.
- tofinish message uses total ordering, however it is a very small message containing a tid value: as no tofinish=false message is sent, a tofinish reception implies tofinish=true.
- The tofinish table can be cleaned up each time a transaction ends.
- bcast(effective_commit(tid)) is protocol dependent.
- abort_conflicting_transactions(tid) aborts all transactions whose result sets conflict with tid's. It has to use a mechanism similar to the one used in [4] in order to rollback conflicting database transactions immediately when required.
- abort_conflicting_transactions(tid) execution before the effective_commit message broadcast is very convenient because the time elapsed between the total order broadcast of (tofinish(tid)) and the broadcast of effective_commit(tid) is not to be underrated; as the last message construction can be time costly and resources consuming.
  An early execution of abort_conflicting_transactions(tid) allows the rest of the nodes to advance work in parallel; specially the abortion of transactions with write operations executed before the committing one operations. These operations will be blocking the database and must be aborted as soon as possible.

4.4 The metadata managers and the transaction metadata

Figure 4.5 shows the architecture elements (a more complete picture was shown in figure 4.1) directly related to metadata maintenance. The MetaDataManager and the Manager are the pieces that populate TransactionMetaData and ObjectMetaData structures:

- The MetaDataManager is the core of all metadata operations.
- The Manager manages the basic metadata set and other straightforward attributes implicit to calls to Manager.

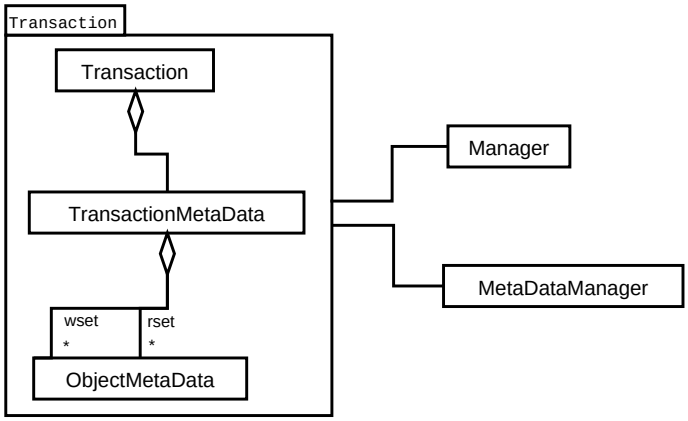

Figure 4.5: Metadata Managers.

When a transaction commits, it collects information, selects the parts needed for each protocol and then these parts are sent to the corresponding transaction metadata objects for them to perform their persistence.

TransactionMetaData objects are then fairly easy software pieces: They fill the metadata attributes for each *oid*.
To describe the way these attributes are collected, we assume an instance pA of protocol A. The remaining components are:

```
Transaction t;
Manager mgr;
ProtocolProxy proxypA;
MetaProtocol mpA;
TransactionMetaData tm;
ObjectMetaData om;
```

4.4.1 Object metadata collection

***Basic type* – oid**

oid fields can be seen in section 3.2.

```
class Manager {
  public String getNewObjectID();
  ...
```

The object identifier is basically a string the system returns that ensures that there will only be one object with this identifier in the whole system. To achieve this, we propose the oid to have the following fields:

- The repository where it is stored and other hierarchical wrapping classes.
- The owner node identifier. For us, it is the node where the transaction that created the object was started.
- A sequential number.

There are other alternatives such as assigning each node a range of possible objects for creation but we find this option to be a better choice because of the ownership concept some protocols use.

We ask the Manager for new objects (INSERT sentences) and once created they are obtained from the TransactionMetaData. With this class we will be able to determine the access_type too.

***Basic type* – info**

Object Info is maintained by the Manager. The manager isolates the protocols from most database peculiarities and one of them is the information required (including dependencies but not its metadata) to update a given object value in the database.

The Manager is able to obtain the ObjectInfo for a set of objects and it is able to receive a set of ObjectInfo objects and update the database.

***Derived type* – owner**

It can be derived from the oid using ObjectMetaData.getOwnerNode(). For protocols that deal with the ownership concept, this value is accessed intensively and then, it is advisable to keep the derived value in memory for successive retrievals.

timestamp

timestamp holds the moment an object was accessed:

```
proxypA.objectsAccessed(String[] oids) {
  ...

  mpA.tm.setTimeStamp(oids, System.currentTimeMillis());

  pA.objectsAccessed(oids);
  ...
```

This code can be described simply as:

```
@before pA.setTimeStamp(oids):
  mpA.tm.setTimeStamp(oids, System.currentTimeMillis());
```

version

When an object is inserted, version is set to one and any time it is updated, it is incremented. If increments were always one unit increments we could the the Manager manage this attribute without help but this is not always true. Think, for example, about lazy protocols where some nodes don't participate in certain transactions execution: This means that some nodes miss some updates.

Our approach for version management implies that versions have to be broadcast where necessary. For protocols using locks it is included when requesting the lock because the request is done for a certain version. For other protocols using versions the message allowing to commit includes the version values. The same applies for the remaining protocols, for them versions are piggybacked.

lock_node

Granting locks are internal operations invoked during the Manager request for commit. This means that we cannot use the proxies to obtain the metadata.
The protocols that need this attribute are **COLU** and **FOB** and they need a round of lock requests between the system nodes.

Assume we have an operation for all the protocols that deal with locks:

```
grantLocks(String[]oids, LockType l);
```

Where LockType $\in \{sread, read, write\}$.

Remember that lock_node exists only inside a transaction. Once it is finished, its value is useless. As only these two protocols use it and the process to obtain it is exactly the same, lock_node does not difficult a change between these two protocols.
In a change from **COLU** or **FOB** to another protocol we will simply discard its value. On the other hand, for a change from a protocol to **COLU** or **FOB** lock_node is useful when the transaction aborts (and it needs to release locks). This is easily solved sending the release message to all nodes whether the previous protocol dealt with locks or not (in this latter case, the message is discarded).

apply_into

This value is known when tid.getNodeOwner() node knows for sure that the transaction commits and commands the others to commit too.

```
@before pA.send(destinations, commit):
  mpA.tm.setApply_Into(destinations);
```

sites

The sites where an object exists is initially a predefined metadata attribute and each time an object is created in a node with an INSERT operation, the list is updated.

threshold

Once the timestamp value is obtained, **COLU** needs to calculate the probability of having these objects up-to-date and compares them with a the global adaptive threshold.

```
@before pA.objectsAccessed(oids):
  mpA.tm.setTimeStamp(oids, System.currentTimeMillis());
  mpA.tm.setThresholds(oids);
```

If pA has an instance of **COLU**, `pA.objectsAccessed()` will use the threshold values.

Qrw

q_r and q_w are obtained combining the information offered by sites and the current view nodes.
The Membership protocol event *change-view* calls the metadata instance in order to update q values.

Arw

The same that applies for the maintenance of q values applies for A_{rw} values.

4.4.2 Transaction metadata collection

***Basic type* – tid**

Method signature at Manager:

```
class TransactionMetaData {
public String getTransactionID();
...
```

The transaction identifier holds a format similar to the oid one. As well as a sequential number, it holds the repository that the transaction is accessing and the node where the transaction was created.

***Basic type* – rset, wset and srset (access_type)**

The Manager has to be able to determine which kind of access an object is having from each transaction. According to this kind, it will include the object in the read set, write set or short read set from the corresponding TransactionMetaData.

Besides updating these lists, the Manager will inform the underlying protocol about the accesses. The protocol can obtain then the lists from the TransactionMetaData structure.

***Basic type* – execution**

The attribute execution is maintained by the Manager. It holds the list of SQL sentences the user has executed. The consistency protocols broadcast this information in order for the other nodes to apply the same set of operations (with the same order) everywhere in the system when the transaction *commits*.

BOT, EOT and BOC

Operations to create and close transactions are not left to the protocol developer but they are part of the Metaprotocol facilities that developers are encouraged to use.
Thus, these createTransaction() and closeTransaction() operations reside in the Manager and contain timestamp retrieval and set the BOT and EOT fields. Actually:

- BOT → Manager.createTransaction() is called.
  If the protocol *Change Points* are the beginning and the end of the transaction then, as BOT duration is a transaction, there is no problem having two BOT and $TS(t)$ formats. Otherwise both values must be maintained: BOT is immediately obtained. $TS(t)$ requires piggy-backing messages among different nodes.

  ```
  // Constructor
  public TransactionMetaData tm() {
    bot = System.currentTimeMillis();
  }
  ```

- EOT: Immediately after the actual database call to commit.
- BOC (**Def** or **Imm**): Begin of commit.

readfrom

Transaction t obtained its accessed objects values from readfrom transactions. The Manager component is the single piece that is able to know this information so it will complete this information.

executed

Its default value is false and only changes when the commit request is done and the transaction is not yet to-delivered.

```
@after pA.commit():
  mp.tm.executed = true;
```

commitable

Set when the transaction is to-delivered in **Jim**, [18].

abort

abort is known when the protocol decides the result of the transaction

```
@after pA.commit():
  mp.tm.abort = pA.commit();
```

Chapter 5

Summary and further considerations

This work has been addressed in different incremental ways. First, we have started explaining in chapters 2 and, optionally, in appendix A how a protocol exchange can be carried out. Once this was settled we undertook the task of the multi-protocol environment definition and design. For that means we needed a set of protocols and their definition. We have them defined through their metadata in chapter 3. Then we needed a common architecture, shown in chapter 4. Finally we presented in section 4.3.1 the core of the metaprotocol.

Chapter 4 pursues an architecture for multi-protocol management in general regardless of the replication model they follow. Section 4.1 thoughts about the metadata structures imply that selecting a unique replication model would simplify the basic metadata structures[1] and would unify the way updates are broadcast. This would be a good incremental strategy in order to simplify the implementation complexity in its first phases.

Recall that our goal (section 1.1) is to obtain a light and efficient metaprotocol. We find these properties to be fundamental as we have focused our success on achieving minimum overhead for the independent consistency protocols implementation and behaviour in normal conditions.

This same section 1.1 in the introductory chapter also focused on the technical motivation of our work. Now let us consider some cultural motivations.

Companies reluctance to adopt replication is due to several reasonable concerns. One of them is redundancy and another one is the impression of loss of control over the data scope. These concerns limit most replication approaches to a rack of clusters inside the same data center. This approach is certainly necessary and increases availability but, while replication over a WAN would offer countless advantages, cluster replication still limits scalability through geographically scattered locations heavily under network bandwidth and it imposes restrictions and operational procedures (such as nightly batch processes or data inconsistency allowance during certain intervals) to achieve data consolidation over several data centers that usually interrupt or unbecome the regular service.
An organisation with such a problem is to be a large organisation and, certainly, no single consistency protocol is able to manage efficiently all its information systems isolation and service level agreement needs.
While the studied protocols solve the redundancy control problem, our metaprotocol allows protocol election and exchange and, due to that, it is a very serious reason to reconsider any previous reluctance towards replication.

[1] Specially the TStamp data type from TransactionMetaData and ObjectMetaData.

Regarding the concern about data control loss, a WAN is not necessarily the Internet. Most companies have designed private WANs and control privacy inside their global environment with all the available physical and logical devices. Even though our design can easily include encryption for the data sent though the network we have not considered privacy as an issue of the metaprotocol itself because the network infrastructure (firewalls, secure communications, etc.), the hardware infrastructure (data center security and operating system security), the application design (responsibilities oriented design, connections encryption, appropriate database roles and permissions and appropriate application user profiles) and the company privacy procedures (privacy assurance contracts, restricted area accesses, etc.) are more appropriate places to solve the problem.

Letting information flow through all the nodes obviously requires a certain degree of organisation but not allowing it imposes a serious burden to a company. Consider that geographical diversity is usually imposed by different area business needs. These needs lead to different data exploitation requirements. Being able to share a part of the data while other parts are still private (either because there are mechanisms to guarantee it or because this part was not replicated) makes unnecessary most consolidation processes because replication is transparent.

For all the reasons exposed a system capable to hold multiple concurrent protocols is necessary and our architecture solves this environment in a simple manner. Multiple protocols are useful for information systems with many different applications and protocol exchange fits well for simpler environments.
Another potential use of the protocol exchange would be to maintain two exchangeable versions of each protocol: One fast for non-critical applications or for academical purposes and the other secure for commercial purposes including mechanisms such as protocol queues serialization, more complete failure models and cache techniques for memory and processor usage minimization.

Appendix A

Sequential protocol exchange

Even though sequential protocol exchange is not the goal of this study, it was the first step we took in order to approach the rest of the work.

This section explains how start an change protocol operations could turn the system to a mono-protocol environment with exchange capabilities.

A.1 Start protocol

Starting a protocol is introduced in section 2.2.

In order to prevent from installing several protocols at the same time we will first justify the use of a install_new_protocol message.
If it was not used then the first message for each protocol should provide any means for protocol election decision. Whatever decision is taken one of the protocols would be installed while the others would not. Transactions using the not-installed protocol cannot be restarted because we cannot assume that the sent message admits a negative acknowledgement, they have to be aborted. The negative acknowledgement has to include indications about the protocol to be installed and this may lead to constant disagreement (see figure A.1).

This reiterative disagreement can be avoided by broadcasting in total order the first message, however it still leads to one round of one transaction abortion for each of the nodes that choose the loosing protocols[1].

These are not an efficient or elegant solutions and we conclude that it is better to use a 2PC algorithm for protocol installation in order to inform all nodes about the chosen protocol so that they can answer with a confirmation or a rejection. We will now describe two approaches for the *install_new_protocol* process: first–one–wins and voting. Both, especially the voting one, allow nodes to be silent and to decide not to give any preference and then to follow indications from the other nodes.

A.1.1 First protocol installer wins

```
void installProtocol(ProtocolID pid) {
 if (for all i in nodes stopped[i] == STOPPED)
   totalOrderBcast(install_new_protocol(pid));
}
```

[1]These nodes are the ones whose transaction first message is ordered behind the first one ordered by the total order.

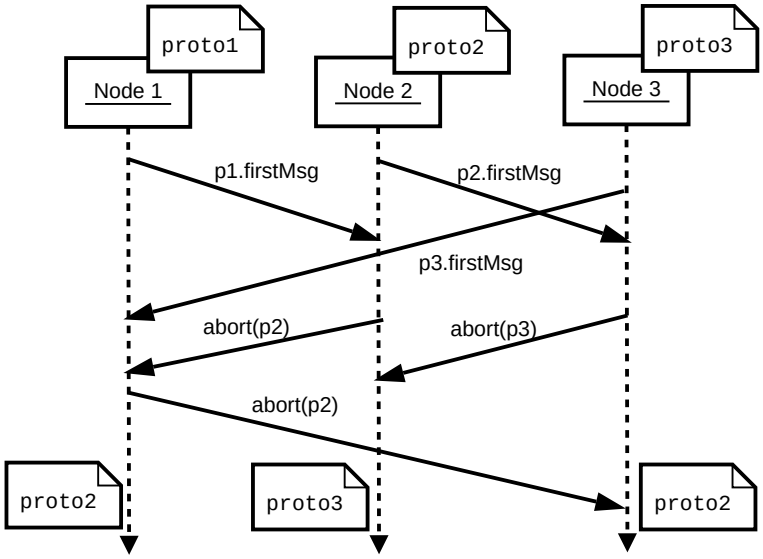

Figure A.1: Start protocol without explicit message.

The previous method starts a protocol installation. No installation is started if a protocol is already installed and the command is broadcast using a total order service (see [14]). For node failure cases, adding uniformity to the broadcast delivery guarantees will do.

```
void receiveInstall(install_new_protocol inst) {
  if (stopped[localID] == STOPPED)
    startProtocol(inst.pid);

  // else discard message
}
```

Once a node receives an install_new_protocol it installs the protocol if the node is stopped or it discards the message if a previous install_new_protocol has already arrived. startProtocol() installs the protocol and changes the stopped flags.

This is an easy protocol but the total order service is costly. One way to overcome this situation is to choose a coordinator node. This way the installProtocol(ProtocolID pid) method substitutes the broadcast with a *unicast* invocation:

```
void installProtocol(ProtocolID pid) {
  if (for all i in nodes stopped[i] == STOPPED)
    send(install_new_protocol(pid), coordinator);
}
```

The receiveInstall(install_new_protocol inst) method changes to:

```
void receiveInstall(install_new_protocol inst) {
  // if stopped install protocol (and change flag:
  if (stopped[locaID] == STOPPED)
    startProtocol(inst.pid);
  // if the node is not stopped discard message in any case:
```

```
   else return;

   // broadcast an installation request:
   if (localID is coordinator)
     bcast(inst);
 }
```

Once an install_new_protocol arrives to an stopped node the protocol is installed. If the node has already installed a protocol then the message is simply discarded. If the node is the coordinator, it sends the message to the rest of nodes[2].

One drawback for this protocol is that the whole procedure has to be restarted if the coordinator node crashes. For this means, every node that sends an install_new_protocol message to the coordinator writes this fact down. If the coordinator crashes and the stopped flag is still set to STOPPED then the install_new_protocol message is sent again to the newly elected coordinator.
Coordinator election must be performed independently in each node in a deterministic way. One easy way to achieve this is to elect the alive node with the lowest node identifier.

A.1.2 Voting installing protocol

As in the previous installation approach, a voting protocol can be designed using *bcast* based procedure or a coordinator based one. We discard the pure coordinator based solution here because voting needs an interval for votes collection and we allow nodes not to send any vote at all. This means that we cannot expect abstention (install_new_protocol(null)) messages either because sooner or later[3] a vote could be sent.

For the same reason, a total order delivery broadcast based solution will lead to a first–one–wins approach. Therefore we will use a *bcast()* message and we will use the lapse of time that the first *bcast()* needs to send to all nodes to collect the votes. The resulting protocol uses a mixed approach because, even though that *bcast()* is used for the votes, coordinators are used:

1. The last node finishes the voting period.
2. The first node sends the install request.

At first, when a node wants to install a new protocol, a vote is broadcast:

```
void installProtocol(ProtocolID pid) {
  if (for all i in nodes stopped[i] == STOPPED)
    bcast(vote(pid));
}
```

Votes are written down in a vector structure such as the one used to hold the stopping process information (stopped):

```
void receiveVote(vote v, int origin) {
  // 1. Write down vote:
  vote[origin] = v.pid;
```

[2]Notice that it is not necessary to send it to the coordinator again: bcast would be better replaced by a multicast where all the nodes but the coordinator are included as destinations.

[3]For example, when the installation requires human interaction.

```
  // 2. If the current one is the last system node and this is the
  // end of the first bcast vote process, notify the coordinator:
  NodeID lastID = highestID(view(v));
  if (localID == lastID && numVotes(vote) == 1) {
    NodeID firstID = lowestID(currentView);
    send(voting_done, firstID);
  }
}
```

Furthermore, for the last system node, the second part of the method sends an unicast message to the first node if this is the first end of a broadcast that it receives.

This approach is very sensitive to system view changes and code needs to cover more availability cases:

1. Newly incorporated nodes in $view_{n+1}$ view are allowed to participate in the voting process but they have to wait until $view_n$'s firstID sends the current voting information to them before broadcasting their votes.

2. Considerations for new nodes:

   (a) lastID is obtained for the view when the vote was sent, $view_v$, because we need to know the scope of the broadcast message. This is due to the fact that if more nodes over lastID join the system during the sending process they won't receive the vote.

   (b) Sending information to $view_{n+1}$'s new nodes is necessary because if one of these nodes is below $view_n$'s firstID it will rule the voting process. As we assume a majority partition model, if they didn't receive this information the new firstID node would take a decision without most of the votes.

3. Considerations for node failures:

   (a) If v_n's lastID node fails, the previous one in v_{n+1} has to send voting_done again.

   (b) If v_n's firstID node fails and voting_done has already been sent, it is sent again to the new coordinator.

   (c) It may be convenient to discuss whether votes from the crashed nodes should be taken into consideration or not.

Once the first voting broadcast is done, the protocol with more votes is requested for installation in all nodes in the current view:

```
void receiveVotingDone(voting_done vdone) {
  // votesDone is a global variable
  votes_done++;

  if (votes_done == 1) {
    // getMax() collects the element that appears more times
    // inside a given vector
    ProtocolID winner = getMax(vote);

    bcast(install_new_protocol(winner);
  }
}
```

An installation message can only arrive from one of the nodes so, when received, it is simply applied:

```
void receiveInstall(install_protocol inst) {
  if (stopped[localID] == STOPPED)
    startProtocol(inst.pid);
}
```

A.2 Change protocol

Changing a protocol was introduced in chapter 2. A sequential change simply needs:

- Stop protocol.
- Sequential start protocol.

Bibliography

[1] APPIA: http://appia.di.fc.ul.pt.

[2] GlobData: http://www.iti.upv.es/groups/sidi/projects/GlobData/.

[3] The ISIS project: http://simon.cs.cornell.edu/Info/Projects/ISIS.

[4] MADIS: http://www.iti.upv.es/groups/sidi/projects/madis/.

[5] Spread: http://www.spread.org.

[6] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. Database Syst.*, 14(2):264–290, 1989.

[7] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of IEEE Int. Conf. on Data Engineering, San Diego, CA, USA*, pages 67–78, 2000.

[8] D. Agrawal, A. El Abbadi, and R.C. Steinke. Epidemic algorithms in replicated databases. In *PODS '97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172, New York, NY, USA, 1997. ACM Press.

[9] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[10] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec. San Jose, CA USA*, 24(2):1–10, 1995.

[11] J. M. Bernabé-Gisbert, Raúl Salinas-Monteagudo, Luis Irún-Briz, and F. D. Muñoz-Escoí. Managing multiple isolation levels in middleware database replication protocols. In *Lecture Notes in Computer Science*, volume 4330, pages 511–523. 4th International Symposium on Parallel and Distributed Processing and Applications, Sorrento, Italy, Springer. ISSN 0302-9743, Dec 2006.

[12] K. Birman, A. Schiper, and P. Stephenson. Ligthweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug 1991.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM Press.

[14] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. *S.J. Mullender editor Distributed Systems (2nd Ed.)*, pages 97–145, 1993.

[15] L. Irún-Briz, F. Castro-Company, H. Decker, and F. D. Muñoz-Escoí. An analytical design of a practical replication protocol for distributed systems. In *Lecture Notes in Computer Science*, volume 3236, pages 248–261. FORTE 2004 Workshops The FormEMC, EPEW, ITM Toledo, Spain, Oct 2004.

[16] L. Irún-Briz, F. D. Muñoz-Escoí, and J.M. Bernabéu-Aubán. An improved optimistic and fault-tolerant replication protocol. In *Lecture Notes in Computer Science*, volume 2822, pages 188–200. Proc. of 3rd Workshop on Databases in Networked Information Systems, Aizu, Japan, Springer, Sept 2003.

[17] R. Jiménez, M. Patiño, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, Sept 2003.

[18] R. Jiménez, M. Patiño, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 477, Washington, DC, USA, 2002. IEEE Computer Society.

[19] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. How to select a replication protocol according to scalability, availability and communication overhead. In *SRDS*, volume 00, page 0024, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[20] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *International Conference on Distributed Computing Systems*, pages 156–163, 1998.

[21] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, Sept 2000.

[22] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware base data replication providing snapshot isolation. In *ACM SIGMOD*, pages 419–430, New York, NY, USA, Jun 2005. ACM Press.

[23] F. Muñoz-Escoí, L. Irún-Briz, P. Galdámez, J. Bernabéu-Aubánand J. Bataller, and M. Bañuls. Globdata: Consistency protocols for replicated databases. In *IEEE-YUFORIC'2001 Valencia, Spain*, pages 97–104, Nov 2001.

[24] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 126–137, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[25] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pages 206–215, Nürenberg, Germany, Oct 2000. IEEE Computer Society.

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 264–274, Taipei, Taiwan, R.O.C., Apr 2000. IEEE Computer Society Technical Commitee on Distributed Processing.

[27] M. Wiesmann and A. Schiper. Replication techniques based on total order broadcast. *IEEE Transactions on Knowledge Data Engineering*, 17(4):551–566, Apr 2005.

[28] ANSI X3.135-1992. *American National Standard for Information Systems – database language SQL*. Nov 1992.