University of Edinburgh

Department of Computer Science

Analysis of a Distributed Commit Protocol for Internet Commerce

M.Sc. Project Report

Tanja Schulz

September 16, 1997

Abstract

In a distributed computing environment, a commit protocol is needed to ensure that a distributed transaction, affecting several sites, either commits at all sites or aborts at all sites; this atomicity of action should happen even though failures occur.
New commit protocols are assessed by their efficiency during normal operation and by their robustness in the face of failures. For complex protocols, such an assessment can be carried out by a combination of detailed analysis and simulation.
In the present project we examine a new distributed commit protocol, the Transaction Internet Protocol (TIP), that is intended for use in Internet and intranet commerce.
An internet environment poses special requirements on such a protocol because of the diversity of communications protocols, greater security concerns and less reliable network latencies.
The results of this analysis suggest that TIP may be satisfactory for simple cases, but its utility for more complex transaction processing is doubtful.

Acknowledgement

I would like to thank Dr. Peter Thanisch for his constant and constructive support throughout the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In distributed environments, a process may need to access data on physically distributed computers/sites, each of which is an autonomous database server. When updates have to be done at two or more sites it has to be ensured that either none or all of the required changes will be made permanent at all of the participating sites. Transactions are used to aggregate individual modifications to a database into a single atomic modification that appears to occur either entirely in a single moment or not at all and Commit Protocols define the rules that the participants follow to reach consensus about whether to abort or commit a transaction ([8]). A transaction may be defined as a 'collection of operations on the physical and abstract application state' ([10]). This means that a transaction is described by all effects that the execution of an operation has on a database.

In many conventional protocols a coordinator is defined for each transaction. The coordinator is responsible to control the correct usage of the protocol, to get all necessary information from the participants and to inform all participants about the outcome of the protocol (whether to abort or commit).

Two circumstances conspire to make commit protocols complicated. Firstly, database systems are typically multiuser and locking is used to prevent interference. Secondly, failures, such as site crashes and network outages, can occur. The commit protocol for one transaction might be stalled because of a failure and this means that all participating sites must retain locks on its data items. This could cause blocking and even deadlocks. Blocking occurs if a coordinator fails and the operational sites cannot determine the final outcome of the protocol.

Therefore special procedures have to be defined to handle failures. Some commit protocols define special termination protocols for cases in which a coordinator fails. A protocol may allow a new coordinator to be elected and the transaction processing may go on.

In traditional distributed environments the participating sites usually belong to one company or companies with a very close relationship. The environment normally uses the same or compatible components (Transaction Manager, Network, Protocols) and they are all adjusted to each other. Another characteristic of such an environment is that the access pattern to the data on different servers is quite predictable.

As a medium for transaction processing, the Internet presents a radically different environment.

According to a survey performed by META Group Consulting, Internet-based electronic commerce will become increasingly important over the next years. They project that in the year 2000 more than 300 billion US$ in Internet-based electronic commerce activity will transact over the Internet and that electronic commerce activity will make up an increasingly important part of overall trading and purchasing activities ([9]).

Due to the changing environment, the requirements on a distributed commit protocol change as well. One of the big advantages of Internet is its flexibility. As a consequence, the environment is much more unpredictable and heterogeneous than traditional transaction processing environments. The distributed transactions will affect the databases of more than one company and issues like security and blocking become more important. No company wants its system to be blocked because of a failure in the system of another company.

At the moment there exist several proposals to address these special problems, including OSI TP ([11]), CORBA OTS ([12]) and TIP ([1]).

TIP (Transaction Internet Protocol) is a distributed commit protocol published by Microsoft and Tandem as an Internet draft. It defines an approach that makes the commit protocol independent of the communication protocol used. It is intended that TIP will be adopted by the Internet Engineering Task Force (IETF) as an Internet standard.

This project is concerned with the evaluation of TIP.

It comprises a theoretical evaluation and a simulation, aiming at understanding the special requirements posed on a distributed commit protocol in an unreliable environment, analysing how the proposed protocol meets these requirements and define its strengths and weaknesses.

The project splits up in two main parts:

1. analysing and understanding the proposed protocol as well as defining the weak spots
2. the implementation in a simulation package.

The complexity of the protocol requires a simulation tool to gain insight into the protocol and how it behaves by evaluating possible events and failure scenarios.

The specification of TIP that has been published in the Internet ([1]) was the basis for my work. Additionally, a TIP mailing list exists in which the features of, and possible amendments to, the protocol are discussed. This mailing list has been very active throughout the duration of the project and the authors of the draft have been developing and refining the draft document in response to comments from the community. This information had to be considered for the evaluation of the protocol as well.
Besides the draft of the protocol, a reference implementation of the first (i.e. obsolete) version of the protocol (written in Java) had been published. Using these two main documents, I spent the first weeks trying to understand how the protocol works.

There are two main reasons why TIP is different from earlier distributed commit protocols. Firstly, it is intended for use in a heterogeneous distributed environment where each site is controlled by a different enterprise. This means that it must be possible to use the commit protocol over a diversity of networks, communication protocols and applications. A restriction to predefined environments would restrict the usability of the protocol and not support the flexibility of an Internet environment.
Secondly, TIP is intended to reflect the browsing style that typifies Internet usage: the transaction grows as the user flits from site to site and the linkage, for commit purposes, follows the same pattern. To achieve this, the protocol must provide a flexible way of generating the transaction tree.

After I obtained a good enough understanding of the protocol, I started to implement it in the simulation environment. The use of simulation in this project is unconventional because we are not interested in measuring simulated time. In a distributed environment too many different factors influence the amount of wall-clock time that will be needed to complete a protocol. We are interested in simulating the behaviour of the protocol, using message passing as the performance metric.

A software that simulates some aspects of traditional 2 and 3 PC protocols was developed by Steven Ramsay in his 4th Year Project ([14]). Although it was initially hoped that we could extend that software to model TIP, it became apparent that TIP differs radically from traditional protocols in what needs to be simulated. The main differences between the traditional 2PC protocol and TIP are that TIP is a decentralised protocol, i.e. there is no central coordinator, and that TIP includes commands to grow the transaction tree dynamically whereas with traditional 2PC, the topology is predefined. Due to these major differences, much of the original software had to be re-designed,

but allowing the new version of the simulation software to perform a comparison between the different protocols.
The software has been written in Java, a relatively young programming language, that is still undergoing basic upgrades.

For the actual simulation and animation of the protocols, other Java-written software packages, namely SimJava and SimAnim ([13]), have been used.

Originally it was intended to use the reference implementation that has been published by Microsoft and Tandem as the basis for the simulation. Due to the fact that this implementation had been written using the first draft of the protocol and that the changes to the second version have been quite basic, this was not feasible.

The last step of the project was to define possible failure scenarios, simulate them and evaluate the behaviour of the protocol under these circumstances.

The content of the rest of this report is as follows: In Chapter 2 I will explain the general functionality of TIP and its importance.
Chapter 3 contains a description of the problems that became apparent during the analysis and in Chapter 4 the implementation of the protocol into a simulation environment is described.
Chapter 5 describes the evaluated scenarios and comprises the results of the simulations. In Chapter 6 general conclusions are expressed.

Chapter 2

Transaction Internet Protocol

2.1 Distributed Commit Protocols

In distributed environments, transactions are used to aggregate single modifications to a database into a single atomic modification. this atomic modification ensures that updates are performed either entirely or not at all. Special protocols have been defined to allow physically distributed sites to reach consensus about whether to abort or commit a transaction.

Existing standards for distributed commit protocols usually employ a 'one-pipe' model in which messages associated with the transaction protocol flow over the same conduit (pipe) as the application to application communications. The transaction tree is built implicitly as an application component communicates with another application component. This transaction tree defines the way in which different sites are included in a transaction and communicate with each other.

To achieve the implicit generation of this tree, the transaction protocol has been embedded within the communications services provided to the application (usually by a Transaction Processing Monitor (TPM) - see [7]).

The implementation and maintenance of such a Transaction Processing Monitor is expensive (in terms of money, time and expertise !) as it is a very complex system. Especially small companies can often not afford this and do not even need all of the offered facilities. They need a lightweight system that is easy to understand, implement and maintain.

Another drawback of the traditional protocols is that the application may only use the communication services supported by the transaction protocol. That implies that for each new communications protocol a new transaction protocol has to be developed.

Consequently, a product must also implement the communications protocol in order to be able to support the transaction protocol. This increases the complexity which in turn inhibits implementation. Instead of a wide implementation of the existing standards, numerous proprietary transaction protocols have been invented. The big disadvantage of these protocols is that they only work between themselves. This is wasteful and restricts inter-operability. The effect is that transactions may only be used where each component of a distributed application is running under the same TP Monitor software. Therefore it must always be known in advance which application components are to be involved in a transaction.

This predictability does not match the flexibility that people have come to expect from Internet usage. A protocol that is suitable for Internet commerce should be a standard transaction propagation and coordination protocol that works independently of the application to application communication mechanism. The two-pipe model adopted for TIP is a model of separation of transaction control and application communications which is used by many existing proprietary transaction protocols.

TIP ([1]) has been designed for achieving atomic commitments in an Internet environment. A typical scenario for the use of a protocol in such an environment would be a user who browses the web and wants to purchase goods from several vendors at different web sites. This user wants either all of the goods or none. The commit protocol has to be able to handle this flexible generation of the transaction tree, allowing all requested servers to be included even if different platforms (e.g. networks) are used.

TIP is based on the standard 2 phase commit (2PC) protocol and employs the two-pipe model. It achieves its simplicity by specifying only how different nodes agree on the outcome of a transaction. The applications have to communicate via other protocols. Thereby issues like application communication semantics and data representation can be avoided.

TIP provides two major innovations compared to current standard protocols:

- a PULL method to propagate a transaction from one site to another (a new participant requests an existing one to add it to the transaction). This allows a client to access an arbitrary set of servers and include them within the same transaction.
- a method where transaction demarcation is controlled by the client application, but the client node does not participate in transaction commitment and recovery. This obviates the need for a full TP Monitor implementation on the client system and is most appropriate for internet applications where a client is typically running on a desktop machine using a web browser with no local recoverable resources.

The benefits claimed for TIP, as opposed to existing transaction commit protocols, are:

- the application may choose whatever protocol mechanism (e.g. HTTP) is most appropriate to communicate with a partner.
- there is no predetermination required regarding which application components are to participate in a transaction. To use TIP the nodes simply have to support a TIP compliant transaction manager (TM) which manages the coordination of local resource managers with remote TIP TMs. The TMs on the different nodes do not have to be the same ! A second requirement is that the cooperating applications agree about their communications data formats and protocol.
- TIP can be used for new communications schemes as well - there is no need to develop and implement a new transaction protocol for each new communications protocol.
- Because of the utility of TIP, a wide implementation is expected. Therefore it will become a practical solution for heterogeneous transactional inter-operability.

TIP is ideally suited to internet transactions where actions of a client are relatively unpredictable regarding which components will comprise a transaction. High performance is supported by allowing multiplexing (multiple transactions use one TCP connection) and pipelining of commands (reduces network latency and resource consumption).

2.2 TIP - how it works

The first step for using TIP is to establish a connection between the participating sites. TIP requires a reliable ordered stream transport, preferably with a low connection setup cost. This is important because generally there will be one connection for each simultaneous transaction that has to be processed (although a multiplexing feature is contained in the protocol design). The connection may be reused for following transactions.

During the course of the protocol, a connection enters a number of different states. The change from one state to another occurs at different points for the site that initiated the connection (=primary) and the site that has been contacted (=secondary). The primary is the only site that is allowed to send commands and changes its state when it receives a message, whereas the secondary can only respond to commands that it received and changes its state after it has sent a message.

The first state that a connection enters, is the INITIAL-state and the primary can send an *IDENTIFY* command to the secondary. The secondary replies with an *IDENTIFIED* command if it is able and willing to accept the connection. With this identification process, the two participating sites agree upon a protocol version that is to be used. The primary also supplies an endpoint identifier, used to re-establish a connection after a failure. If the primary does not supply an endpoint identifier, no *COMMIT* is possible in the 2PC of the protocol The secondary could also reply with an *ERROR* command. Generally, an *ERROR* command can be issued in any state, either by the primary or by the secondary. It indicates that the sender of the *ERROR* command had received a previous command which it considers was either malformed or issued in the wrong state. The connection then enters the ERROR-state and is no longer useful. As this command may be issued in any state and the result is always the same I will not mention it in the following but take it as granted that this possibility exists.

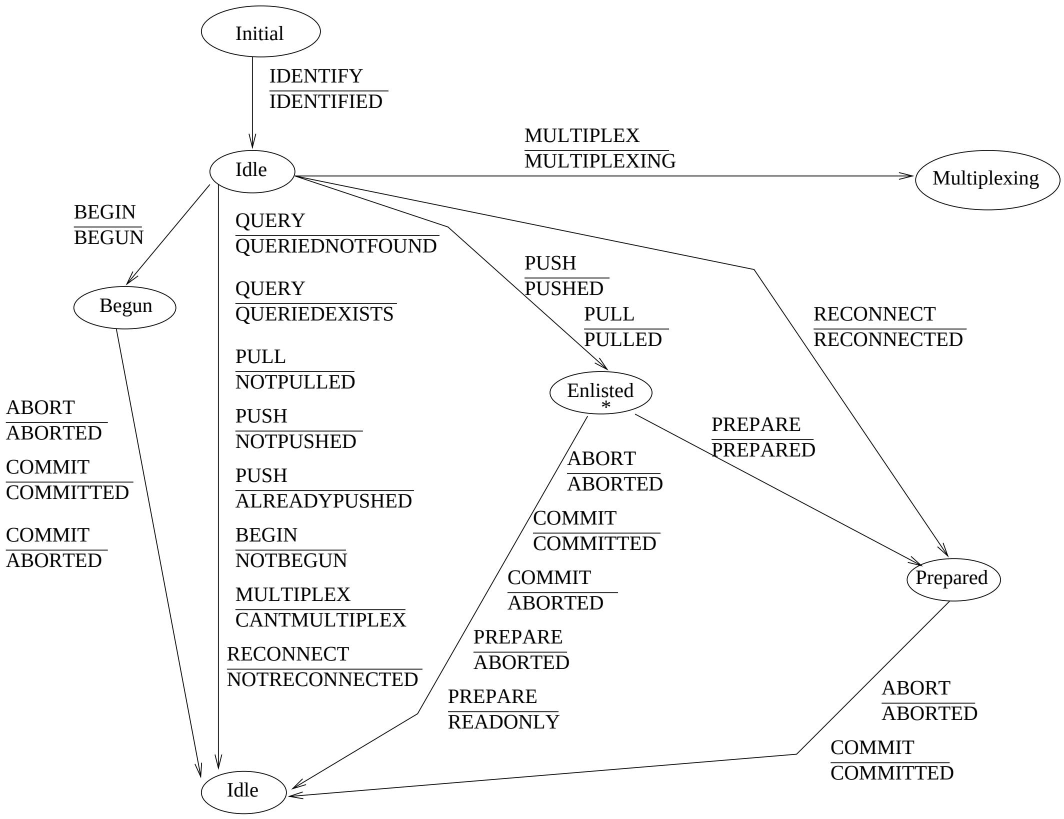
If the secondary replies with an *IDENTIFIED* command, the connection enters the IDLE-state. This is the main state of the protocol because it is the point where a transaction is assigned to a connection. Once a transaction has completed, the connection returns to the IDLE-state and may be used for another transaction.

From the IDLE-state the connection may either enter the BEGUN, ENLISTED, PREPARED or MULTIPLEXING state. Entering the MULTIPLEXING state indicates that the connection will be used by a multiplexing protocol which provides its own set of connections. In this state no TIP commands are possible and the connection can never leave this state.

If the primary starts a 1PC, the connection enters the BEGUN state, otherwise it enters the ENLISTED state. The PREPARED state is entered if a transaction has been prepared for commit. This may either be directly from the ENLISTED state or if the primary and secondary re-establish the connection after recovery from a failure.

TIP allows a number of different commands to be used so that a quite complicated structure of possible message exchanges is generated.

Figure 2.1 shows the possible commands and the change of states. It may be interpreted in two different ways: the command on top of the line may be sent, the one below the line received. In this case the figure shows possible actions and changes of state for a primary. If the commands on top of the line are received and the ones below sent in return, then this figure represents a secondary's actions. *QUERY* and *RECONNECT* are special commands as they are only used after a failure had occurred. The secondary will try to contact the former primary by sending a *QUERY* command whereas the primary will try to re-establish the connection by sending a *RECONNECT* command (see subsection 2.2.2).



* If a PULLED command is issued, the roles change, i.e. the Primary becomes Secondary and vice versa

Figure 2.1: TIP states + commands

The main characteristics of the protocol are described in the following subsections.

2.2.1 Basic commands

From the IDLE-state, the primary may send a *BEGIN* command, indicating that a transaction shall be created in which only one subordinate is involved and there are no local recoverable resources on the primary's site. For this connection a one-phase commit protocol will be applied. This does not exclude the possibility that the subordinate contacts other sites and starts a 2PC with these servers !

If more than two participants are allowed to participate in a transaction, the primary has to *PUSH* or *PULL* the transaction. *PULL*ing can be used if a client started a transaction with one server A and wants to include another server B in the transaction. It obtains a TIP URL including the transaction identifier from A and sends it to B. B contacts server A, asking to enlist in the respective transaction. Server A then knows that it has to involve Server B in the commit process. A TIP URL contains all the information that server B needs to contact server A (host address, port number, transaction identifier). Figure 2.2 shows how the different Transaction Managers and applications communicate to achieve the propagation of the transaction.

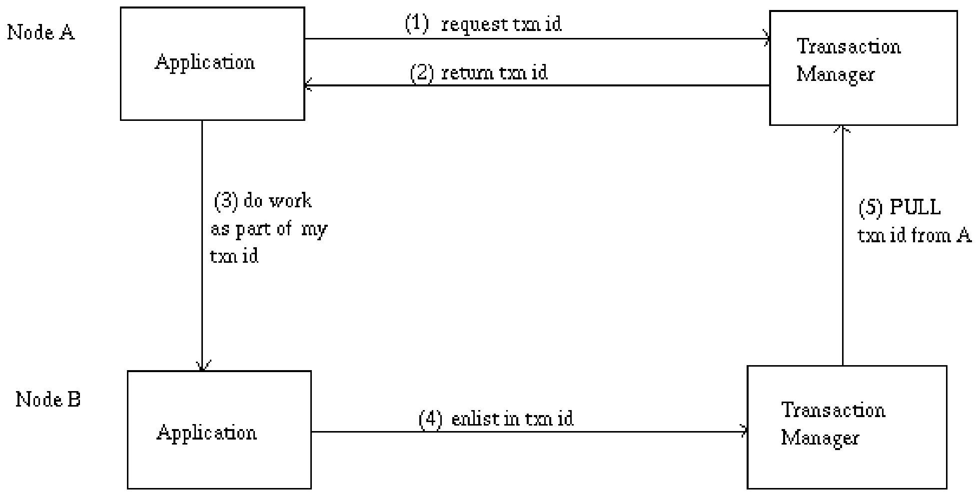


Figure 2.2: PULL

The other possibility to involve new participants in an existing transaction is to *PUSH* it. Figure 2.3 shows the message flow in that case.

PUSH is probably most interesting for cases in which a server A needs to involve another server B in a transaction without the client being aware of it.

The main difference between *PUSH* and *PULL* is that in case of a *PUSH* the Transaction Managers of the two servers exchange the transaction identifier directly, whereas with a *PULL* the application of server B has to tell its Transaction Manager to get in touch with the Transaction Manager of server A.

Once a *PUSH* or *PULL* command has been issued, the subordinate may accept or reject the request. If it rejects it (by sending *NOTPULLED* or *NOTPUSHED*), the connection will enter the IDLE-state and may be used for another transaction or shut down. TIP also allows the secondary to respond with an *ALREADYPUSHED* command if another connection to the same server exists that is concerned with the same transaction. In this case the new connection also

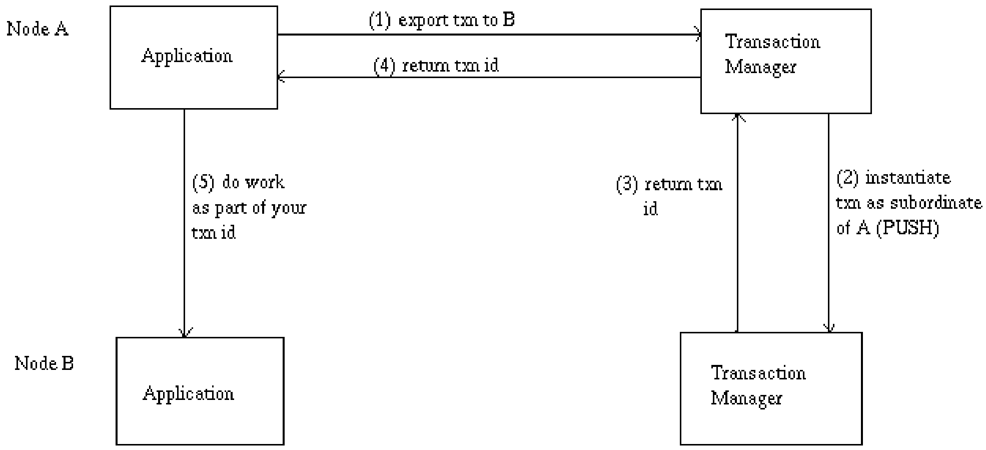


Figure 2.3: PUSH

returns to the IDLE-state.

If the secondary accepts the request (by sending a *PULLED* or *PUSHED* command), the connection enters the ENLISTED-state. In case of *PULL*ing a transaction, this is the point where the roles change. The primary becomes secondary and vice versa. This allows the site where the transaction originated to stay in control over it [1].

In the ENLISTED-state it is possible for the primary to directly issue a *COMMIT* or *ABORT* message. The secondary can respond by sending a *COMMITTED* or *ABORTED* command (see Fig. 2.1) and the connection enters the IDLE-state again (standard 1PC).

To execute the 2PC, the primary sends a *PREPARE* command. The secondary may reply with an *ABORTED* command in which case it aborts the transaction and the connection returns to the IDLE-state.

Another possible response is *PREPARED*. The connection enters the PREPARED-state and the next step for the primary would be to issue an *ABORT* or *COMMIT* command. If the secondary receives an *ABORT* command, it can only reply with an *ABORTED* command and similarly it

[1] In practice, this could be important where one site is more 'trusted' or 'reliable' than another site and is required to have a more central role in the commit.

can only respond *COMMITTED* if it receives a *COMMIT* command. This defines the end of the transaction and the connection will go back to the IDLE-state.

In the PREPARED-state, the *COMMIT* command informs the secondary that the transaction has committed. In contrast to this, a *COMMIT* command issued in the BEGUN or ENLISTED state simply asks the secondary to attempt to commit the transaction.

If no endpoint identifier had been supplied with the *IDENTIFY* command at the beginning, no updates are allowed and the secondary will reply *READONLY* to the *COMMIT* command. The connection will return to the IDLE-state.

TIP requires the applications to control which servers are to be included in a transaction.

2.2.2 Failure

Failures may occur in different states and therefore require different reactions. If a connection fails in the INITIAL- or IDLE-state, no action is necessary because no transaction has been assigned to the connection.

If it fails in the BEGUN- or ENLISTED-state, the transaction will be aborted (presumed-abort 2PC).

A failure in the PREPARED-state is slightly more difficult to handle. If the superior decides to commit the transaction, it must eventually establish a new connection to the subordinate. Once this is done, it sends a *RECONNECT* message. If the subordinate does not know the transaction anymore, it replies with a *NOTRECONNECTED* command. This indicates that no further action is needed because the transaction has been completed successfully (the failure occured after the subordinate had sent *COMMITTED*). The connection remains in the IDLE-state.

If the subordinate replies with a *RECONNECTED* command, the transaction is still in the PREPARED-state on that site and the connection will also enter the PREPARED-state. The superior can then send the *COMMIT* command to which the subordinate has to reply with a *COMMITTED* command. If the superior decides to abort the transaction, it does not have to establish a new connection but may do so. If it does send a *RECONNECT* command and receives a *RECONNECTED* response, it should send an *ABORT* command.

The subordinate on the other hand should periodically attempt to create a new connection to the superior and send a *QUERY* command until it either receives a *QUERIEDNOTFOUND* response from the superior (i.e. the transaction had been aborted before the failure and the subordinate can therefore abort it as well) or it receives a *QUERIEDEXISTS*. If the superior sends a *QUERIEDEXISTS*, it has to issue a *RECONNECT* command at some point. In that case the subordinate will learn about the outcome of the transaction from the superior.

2.2.3 Performance

TIP offers the possibility to multiplex transactions. This means that one TCP connection may be shared by several transactions. The primary sends a *MULTIPLEX* command and suggests a multiplexing protocol that will supply a large number of connections on the existing one. The only defined protocol so far is SCP1.1 (Session Control Protocol, see [4][2]) but other protocols may be defined in the future. The secondary may respond by sending *MULTIPLEXING*. This indicates that it agrees with the suggested protocol, the connection enters the MULTIPLEXING-state and all subsequent communication is as defined by the multiplexing protocol. The connections that are created by the multiplexing protocol start out in the IDLE-state. If a connection fails in the MULTIPLEXING-state, all connections are assumed to have failed and will be treated individually.

If the secondary cannot support the suggested protocol, it may send a *CANTMULTIPLEX* message and the connection remains in the IDLE-state.

The other possibility to improve performance is the usage of pipelining. TIP allows the primary to pipeline certain commands without waiting for the response from the secondary (e.g. *PREPARE* followed by *ABORT*). The secondary has to reply to each of the commands in the same order as they are received.

For more details about the protocol see [1].

[2] This protocol has been renamed to TMP2.0 (TIP Multiplexing Protocol) in the latest version of the TIP protocol specification and is now part of it (see [1]).

Chapter 3

Problems with TIP

TIP is a new protocol which is reasonably advanced and near to becoming a standard. Nevertheless there exists no definitive description of the protocol. This means that a number of issues and requirements are not clearly defined, e.g. what are the possible reasons for a site to send a *NOTPUSHED* command ? What requirements exist for the underlying communication protocol ? What are the requirements posed on the application ?

To understand the protocol properly, a thorough reading of the available documents, discussions on the mailing list and the simulation had to be used in a combination.

As the documents provided are just a draft, some of the below mentioned problems might be solved in a future version.

3.1 Contextual problems

3.1.1 Security

Unlike other protocols, TIP is intended for use where a transaction involves several different enterprises. This gives rise to several security-related problems besides the usual active and passive attacks. Such attacks can be prevented using traditional methods such as encryption, end-to-end authentication and electronic signatures.
TIP may lead to problems where one of the sites participating in the transaction is in fact hostile to the others, but the hostility manifests itself in a subtler way than the traditional active attack.

One of the main characteristics of TIP is that it gives a lot of control and responsibility to the applications. The applications decide which servers to involve in a transaction and how to do it (*BEGIN*, *PUSH* or *PULL*). This means that transactions may be *PUSH*ed or *PULL*ed from a server that the contacted server does not want. A company might use this possibility to influence the reliability of another company's system. E.g., it may *PUSH* a transaction to a server but never complete it. This will cause the other server (or servers) to be blocked or at least occupied with unnecessary work.

Another possibility is that a server joins an existing transaction and causes it to abort although this would not have been necessary (e.g. to prevent that the other company gets the benefit out of that transaction). This may become a problem if it happens regularly and may cause big damages. The solution that the TIP work group proposes to these problems is to use the Secure Socket Layer protocol ([5]) [1] to restrict access to trusted clients and to encrypt messages. The disadvantage of this solution is that it restricts the flexibility of possible connections as it has to be known in advance which clients can be trusted.

Trust is in general a critical point in the way TIP will be used. A server A might trust server B and start a transaction with it. B might contact other servers as well - and this means that A has to trust B also in so far that it uses the same standard for defining trustworthiness as A itself and does not allow any hostile servers to join the transaction.

Repudiation is also one of the problems that must be considered. One enterprise might regret having voted to commit a transaction. For example, a market change might mean that the price is unfavourable to the enterprise and the enterprise might therefore falsely claim it received a message informing it that the global decision was to abort the transaction. The protocol must make it difficult to repudiate the transaction. TIP does not provide a mechanism to ensure non-repudiation. The latest version of the TIP requirements document ([2]) mentions the possibility that such a protocol might be defined later on.

3.1.2 Communication graph

Due to the fact that the application defines the access patterns to the data by issuing *PUSH*es, *PULL*s or *BEGIN*s, different patterns of communication are possible even if the transactions access the same data (depending on where these commands occur).

By sending *PUSH*, *PULL* or *BEGIN* to the different servers that are to be involved in the transaction, the applications define the communication pattern, e.g. if one server is in control of the

[1] This has been replaced by its successor, the Transport Layer Security Protocol (TLS, [6]) in the latest version of the TIP documentation

whole transaction (all other servers *PULL* it from there or it is the only one that *PUSH*es the transaction to another server) or if a chain of servers is generated (one server sends *PUSH*/*PULL* to the next). In the latter case there will be no one site that knows about all involved servers and communication may be subject to long delays. A message has to be sent from the first to the second server which evaluates it, passes it on to the next server and waits for a reply before it can actually send a reply to the first server. In long chains, this may take a considerable amount of time.

Fig. 3.1 shows an example where the application on node A starts a transaction with server B, using a 1PC, and B then includes servers C, D and E by *PUSH*ing the transaction to them.

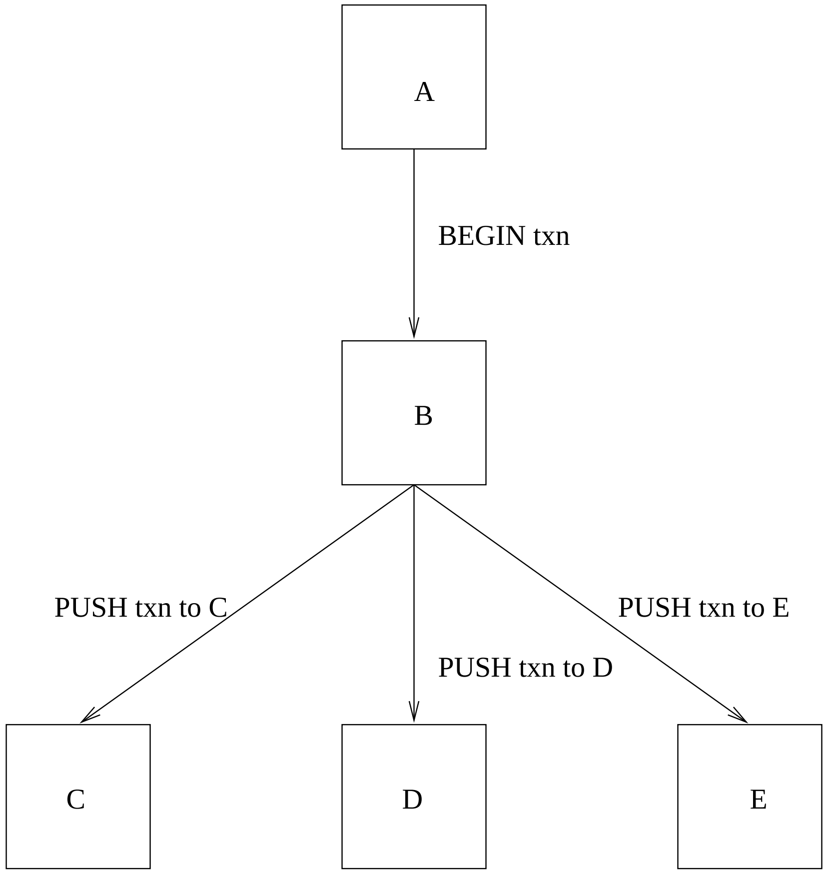


Figure 3.1: Communication pattern using *PUSH*

In Fig. 3.2 the same transaction is shown, this time using a mixture of *PUSH*es and *PULL*s that lead to a different communication pattern.

As TIP does not support a universal transaction identifier (i.e. a txn id which is known to every

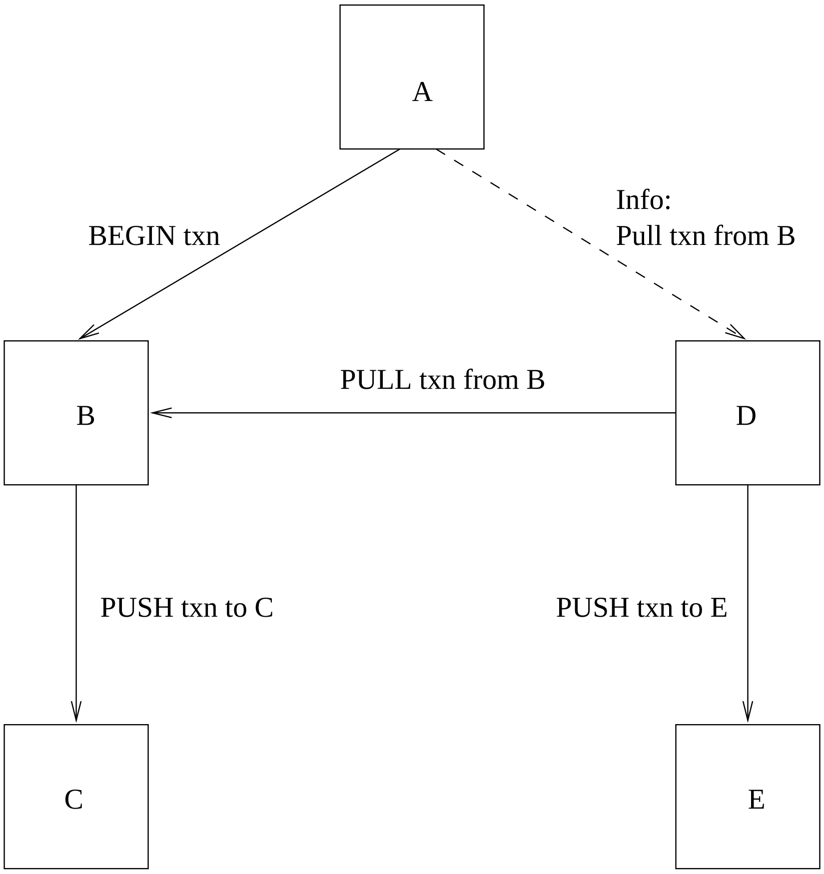


Figure 3.2: Communication pattern using *PUSH* and *PULL*

participating site) and no list of participants is being propagated to all concerned servers, there is no communication possible between the individual participants. This makes the protocol very vulnerable to single-point-failures. If one server in a chain fails, it cannot propagate the decision of a 2PC and all other participants will be blocked. The same problem exists if one server controls the whole transaction. If this one server fails in the PREPARED-state, all participants will have to wait until it recovers. There is no other possibility for them to determine the outcome of the 2PC.

Another drawback is that a server can only ask another server to enlist in an existing transaction at a server where itself is directly linked to. E.g. in fig. 3.1, server E could ask another server F to enlist in a transaction by *PULL*ing it from B but it cannot ask F to join the transaction at A, C OR D because it does not know that they are already involved in the transaction (advantage for E to ask another server not to enlist with itself is that it could avoid coordinator responsibilities)

3.1.3 Generation of Connections

TIP allows a very dynamic generation of new connections. This raises a number of critical issues.

First of all there are two different communication channels (two-pipe model). This allows independence from the communications protocol but on the other hand introduces a problem of synchronisation. For example, a user might tell node B to *PULL* a transaction from node A. If the user then changes his/her mind and decides to finish the transaction without B being involved, A might get confused because of the different messages arriving. A possible message exchange sequence would be like the following:
B receives the message from the user to *PULL* the transaction and sends a request to A. Afterwards it receives a second message, informing it to cancel the first request. At the same time, the user sends a message to A, informing A to finish the transaction. A receives B's request to be enlisted in the transaction before the message that contains the information to commit the transaction. In such a case, A would include B in the transaction although this was not wanted by the user.

The protocol does not define any possibility how B could cancel the request to be enlisted once it did send the *PULL*. Only A could answer *NOTPULLED*, therefore the application has to ensure that all information flows to the correct sites at the correct time. In the above described scenario this would mean that the user should send a message to A telling it NOT to include B rather than sending the message to B.

A similar situation occurs where the user tells B to *PULL* from A and then tells A to commit. A starts to commit before the *PULL* message from B arrives because it does not know that B intends to *PULL* the transaction.

Somehow it has to be ensured that the transaction does not commit/abort before all involved servers are actually included and also that not too many servers are included. On the one hand the application on node A has to ensure that all servers are informed before it requests the transaction to commit/abort, on the other hand all servers have to make sure that they registered all requested sites successfully before they respond to the application.

The communication between TIP and the application has to be defined clearly. It is not clear from the documentation what happens if node B cannot *PULL* the transaction. Most likely it will inform the application on node A about the problem and it has to deal with it. An even worse scenario would be if the application to application communication fails and B can not inform A about any problems. TIP does not care about these kind of problems. This is one of the reasons why it is relatively simple.

3.1.4 Protocol Versions

When initiating a connection, TIP requires the servers to agree on the protocol version that should be used. This is another possible reason for the above described problem because it means that node B's TM might not be able to communicate with node A's TM because they support different protocol versions.

Different protocol versions may also lead to problems in case of failures.

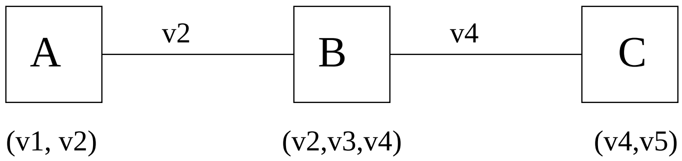


Figure 3.3: Protocol Versions

In Fig. 3.3 Node A supports protocol versions 1 and 2 and agrees with node B to use version 2 as B only supports versions 2 to 4. Node B then initiates a connection to node C on which they will use version 4 of the protocol as node C does not support any earlier version. In such a scenario node A and node C cannot communicate because they do not support a matching protocol version. Despite the fact that A and C do not know that they are both involved in the same transaction, this would be another problem in overcoming failures. If node B fails, node A and C have no possibility to communicate.

3.1.5 Efficiency/Robustness

As already described above, TIP allows servers to be included in a transaction as they are needed. One server might contact another one and this one in turn contacts yet another server. The first server does not have to know about the third server that had been contacted. Therefore a scenario like the following one would be possible: Server A *PUSH*es a transaction to servers B and C. Both of them then include server D in the same transaction (see Fig. 3.4). Server D does not know that both connections are concerned with the same transaction (as no global transaction identifier is used in TIP) and will treat them as completely different ones. If now servers B and C want to access the same data, D will have to abort the transaction on at least one of the connections. The problem is that D cannot grant the necessary locks to both connections, even if one of them just wants to read a data item. Of course this problem cannot be solved by treating the transactions as one global transaction, but it seems to be overly restrictive to treat them as two completely different transactions.

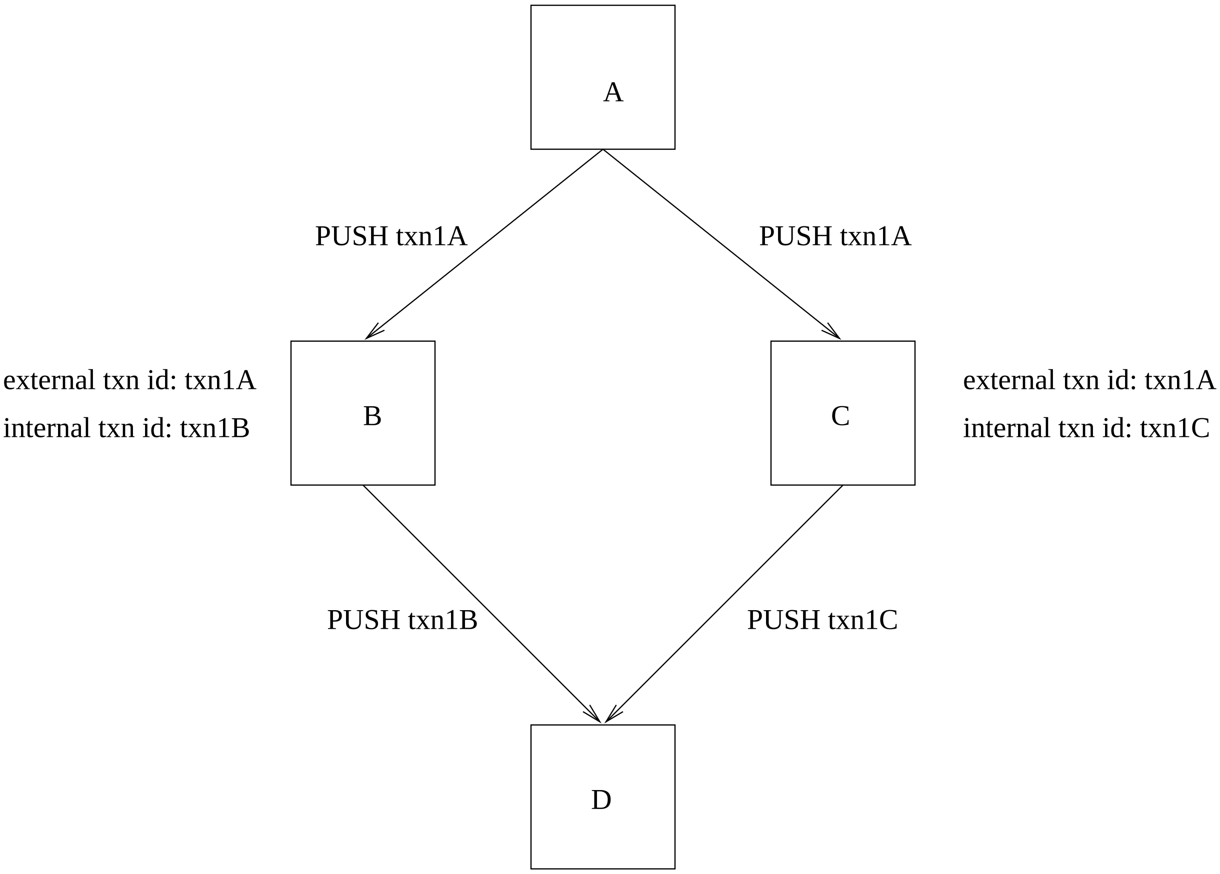


Figure 3.4: Multiple inclusion of a site in one transaction

Another problem that D might face is that it will be contacted by a large number of other servers that are all concerned with the same transaction. This would mean that D has to handle and coordinate all these connections as completely unrelated which causes an overhead that could possibly be reduced if D knew that all these connections are part of the same transaction.

3.1.6 Failure/Recovery

One of the main aims for traditional distributed commit protocols is to make them more robust against failures. One way to achieve this is by making the protocol non-blocking, i.e. use time-outs to discover failures and allow efficient recovery in such a case. TIP does not provide any facilities for an efficient recovery. If the primary of a connection fails in the PREPARED-state, the secondary and all other connections from that site will have to wait until the primary recovers before they can make a decision. The fact that no global transaction identifiers are distributed and that different protocol versions may be used on each connection prohibit that the operational sites get in contact. A third factor that could prohibit communication between two sites is the application-to-application communication. Two sites may be able to use TIP and they even support the same protocol versions, but their applications are not able to communicate because they do not support the same communications protocol.

This is a step backwards with regard to robustness and reliability of a distributed commit protocol.

Another general issue concerned with failures is the way in which they are detected. In traditional distributed environments a failure is detected by a time-out. This concept works fine in a tightly coupled, high-speed network within one enterprise but not in an internet environment. In such an environment different network delays have to be expected and congestions may cause additional delays. The delays that occur are also influenced by the communication tree. It will take longer for a node at the top of the tree to determine if a failure occurred at the end than for a node that is closer to the failed site (i.e. time-out periods add up). The problem is that the top node does not know how many nodes will follow.
Therefore it is very difficult to define an appropriate time-out period.

So far we have only been concerned with failures in the TIP communication. This covers only one part of the 2-pipe model that TIP employs. The other part of the communications is the application-to-application communication. Failures on this level might also cause the protocol to block e.g. because it does not get the information to commit/abort. This might happen because the application that started the transaction did not get all the data it needed due to a failure on that level.

3.2 Internal problems

3.2.1 General problems

One important question is: up to which point in the protocol the transaction may be *PUSH*ed or *PULL*ed ? The documentation provided so far does not give any information about it. If it would be possible to *PUSH/PULL* a transaction although it is already in the PREPARED-state the new participant cannot be involved in the 2PC anymore. But the protocol gives the possibility to directly issue a *COMMIT* or *ABORT* command in the ENLISTED-state.

It is also not clear from the documentation available how the decision is made whether to start a 1 or a 2PC from the ENLISTED-state (send *COMMIT/ABORT* or *PREPARE*). If a primary would start a 1PC with several subordinates, send *COMMIT* to all secondaries and one of them replies with *ABORTED*, then there is no possibility for the primary to inform the other secondaries about this event. Assuming *COMMIT* or *ABORT* can only be sent if there is only one secondary involved in which case no problems would occur. On the other hand there is no need to use *PUSH* or *PULL* if there is only one secondary involved. In that case a *BEGIN* could be issued to start the transaction commit protocol.

Generally it is also not defined how the primary actually contacts the secondaries. Does it send the messages one by one, waiting for a reply from one secondary before it contacts the next one ? Or does it multicast the messages, gather all replies and then decide about the next step ?

These sort of open points are not essential for the protocol but it would ease the understanding if they would be explained more clearly in the documentation.

3.2.2 Message Arrival Order

As TIP is designed to be independent of the underlying communications protocol, it has to be able to handle the behaviour of different protocols. In distributed environments it is possible that the messages do not arrive in the same order as they were sent. They may be forwarded on different routes, with different traffic and delays, e.g. caused by congestion in a router. If one message is forwarded on a different route because the communications system realised that there is a problem on the first route, the first message might arrive after the second one.

The problem with messages that arrive out of order (or if a message gets lost) is that TIP expects commands in certain states. If a message arrives while the connection is in a different state, the protocol always sends an *ERROR* message, enters the ERROR-state and the connection can no longer be used. If the communication protocol does not guarantee that the messages arrive in order, this might cause a lot of transactions to be aborted.

3.2.3 Failure/Recovery

As already mentioned above, there is still a lack of documentation concerning the requirements and implementation of the protocol. This also becomes obvious when dealing with recovery. The available documentation does not provide any information about the log entries that are needed to ensure a consistent recovery.
An attempt to fill this gap has been made in the latest version of the TIP requirements draft ([2]) where possible log-entries have been advised.

Chapter 4

Design of the simulation

The second part of this project is concerned with the simulation of the Transaction Internet Protocol to gain more insight into its behaviour.
Such a simulation is quite different to a conventional simulation because rather than simulating time, we simulate behaviour. The time that is needed to complete a commit protocol in a distributed environment is influenced by many different factors and does not give the information we are interested in. When simulating the protocol, we are more interested in message passing, e.g. number of messages sent, message rounds needed, and how these parameters change in different scenarios (e.g. with or without failures, in case of commit compared to abort).

4.1 Resources

It was decided at the start of the project that the simulation software would be written in Java and would use the SimJava and SimAnim ([13]) simulation and animation packages.

The reference implementation that was available from the Web implemented an earlier version of the protocol. Furthermore, it was not intended for simulation. Rather it was supposed to be the TIP software that would execute at each site. Thus this implementation was not used in the present project.

4.1.1 Java

Java is a relatively new object oriented programming language, based on C and C++.

One of its main advantages is its platform independence which allows the program code to be transferred from one machine to another without having to recompile it. This is possible because a Java program will be compiled into a special architecturally neutral byte-code. The program can then be run on any machine that implements a Java interpreter which takes the byte-code and interprets it at runtime.
This platform independence makes Java especially interesting for Internet programming. A program may be stored on a server and if a client wants to execute it, the byte-code can be copied to the client and run from there.

Java is still undergoing a number of changes. The version used for this project is JDK1.1.3 which might be incompatible with future versions.

4.1.2 SimJava and SimAnim

SimJava is a discrete-event simulation package developed in Java at the University of Edinburgh. It manages event queues and the advance of simulation time to schedule events. This simulation package allows the abstraction from time as it is able to execute without advancing the simulation time.

SimAnim is a package of classes that provide a graphical user interface (GUI) to SimJava simulations. Such a graphical representation of the simulation makes it easier to understand.

4.1.3 DCPsim

DCPsim is a simulation package that simulates the traditional 2 and 3PC protocol ([14]). It allows a user to select the protocol that has to be used for the simulation and to define the specific scenario (e.g. vote to commit/abort, failures). The statistics displayed at the end of the simulation comprise information like protocol used, number of sites involved, number of messages sent and received, number of message rounds, number of participants voting to commit/abort, number and types of failures and recoveries.

The simulation has been written in Java and is designed as an object-oriented software. Separate classes exist for each of the main objects and the behaviour of the different protocols is also defined in individual classes.

The main entities used in this software are:

- Coordinator - controls the protocol processing (i.e. asks for votes, send final decision)
- Participant - site that participates in the commit process
- Filter - coordinates messages that arrive at a site or have to be sent, controls that correct entity receives the message
- Adversary - starts and ends the simulation, controls message passing between the different sites

A site generally consists of a participant and a filter, one site additionally hosts the coordinator.

DCPsim uses SimJava and SimAnim for the simulation and animation of the protocols.

4.2 The implementation of TIP in DCPsim

Although the authors of TIP claim that it is a simple protocol, the implementation was complicated by the major differences between traditional commit protocols and the Transaction Internet Protocol.
One of the main difficulties was that TIP does not have any notion of a coordinator, as it is a decentralised protocol. Furthermore, the behaviour at each site depends on the history of *PUSH*es and *PULL*s. Each node may have more than one connection for a transaction (all of which may be in different states, may make different decisions ...). There is also no static, predictable pattern of accesses - each node may contact any other node. This leads to a very different communication pattern than in traditional protocols and required major changes to the software. A TIP simulation must embody the transaction topology, whereas with traditional 2PC this is not an issue.

4.2.1 Assumptions

To ease the implementation of TIP, a number of simplifying assumptions had to be made.

The main thrust of the project invoked implementing the 2PC part of the protocol. In the experiments the basic assumption was made that no more than 10 sites may be included in a simulation.

In principle, a site could be on the receiving ends of two or more *PUSH*es from the same transaction, but it would not realise this because the transaction identifiers would be different. In our simulation we model this indirectly by treating the second connection as if it were going to a separate site. Each site can only be involved in a transaction once. As a consequence it is not possible to generate cycles for the simulation.

Also, a site can only be involved in one transaction at a time, i.e. all connections for that site are concerned with the same transaction. It is nevertheless possible to have several unconnected trees - there is no definition whether these trees handle the same or different transactions ! The simulation treats such a case as completely independent transaction trees where it does not make any difference if the same or different transactions are involved. This would be the same for a real implementation of the protocol as well because one tree does not know about the existence of the other.

Another assumption was to treat failures as process failures, i.e. failures are defined per connection, not per site. This follows partly on from our assumption that each *PUSH* and *PULL* is to a new site and partly from the fact that only secondary failures have been realised. As it is only possible to have one secondary at a site, only this connection will fail.

4.2.2 Implementation

For the simulation of TIP, some of the basic functions that DCPsim provides have been used. These are mainly the user interface and the integration of SimJava and SimAnim.

The available user interface had to be amended in some points (disable the selection of a coordinator, allow the selection of connections that should be simulated, give the possibility to define one or more specific sites that will abort the transaction).
The possibility to define specific participants that will abort the transaction was required for the TIP simulation. This functionality has also been implemented for the simulation of the traditional protocols to maintain consistency.

One major change to the existing software was that a flexible way of generating the transaction tree had to be introduced. The possible methods that can be used for setting up a connection are hard-coded. The user can select the required connections from a list that contains all possible combinations, combining all sites and all available methods (BEGIN, PUSH, PULL). It is up to the user how many connections shall be generated, but the simulation can only handle one secondary per site, i.e. a site can only be included in a transaction by receiving either one *PUSH* or *PULL* or *BEGIN*, and a site cannot issue more than three *PUSH*es or *PULL*s (*BEGIN* is only allowed with one subordinate !).

This means that for the TIP simulation, a site consists of a filter and max. one secondary and three primaries. However, it will be quite common that different sites host a different number of secondaries and primaries.

Another important amendment was to define separate classes that describe the behaviour of the primary and the secondary. These classes (Primary, Secondary) replace the existing coordinator and participant classes (Coord, Part) in case of simulating the Transaction Internet Protocol.

To define the general behaviour of the protocol a new class (Table_TIP) had to be defined which represents the possible message exchanges of the protocol and its behaviour in case of failures. This class describes all possible actions that a primary or a secondary could take, depending on its actual state [1]. When starting the simulation some basic tables will be set up (e.g possible states, message names, log entries). Upon execution, the threads executing the primary and secondary classes determine their next action by evaluating the possible actions that are defined in the Table_TIP class for the actual state of the entity. If a message has been received, it will be controlled if this message corresponds to the expected message. In case the messages do not match but the table defines other choices for possible actions, these choices will be evaluated. If no choice matches the data that has been received, an error will occur and the simulation will stop. This indicates an error in the implementation (see appendix B for code examples).
The definition of the behaviour of the protocol was quite complicated because the existing documentation does not cover all aspects of the internal processes at one site to coordinate all its connections, e.g. when are the votes from subsequent connections passed on to the superior connections ? When do these connections actually commit ? Only when subsequent connections have reached the end of the transaction or already when they are in the PREPARED-state where the only possible vote is to commit ?

The main difference to the existing software is that the adversary is no longer needed. Due to the many different possible topologies, a central entity that controls all possible message passing is not feasible. Instead, the filter did become much more important. The filter uses a number of the methods offered by the adversary class (e.g. display statistics, method to return the participants intended vote and to set this vote, methods to set the global decision ...) to avoid redundancy.

The main responsibility of the filters at each site are to start up the simulation and to forward the messages between the different connections at the site (e.g. decision to commit or abort).
For the correct forwarding of all messages it is necessary to have a unique identifier for each secondary and primary. This is achieved by combining the site (=filter) number with a number assigned to each connection.

[1] The definition of the behaviour is based on the calculus of communicating systems (CCS, see [15]) although it does not strictly follow its rules.

The filter is responsible to inform all its secondary and the primaries about any decisions taken at the other connections. This is done by exchanging special control messages. Messages between a primary and the respective secondary of that connection are exchanged directly, without involving the filter.

The layout of the animation proved to be slightly tricky and needs to be improved. Although the possible number of sites has been restricted to max. 10 and each site may only comprise one filter, one secondary and up to three primaries there remains a possible maximum of 50 icons that have to be displayed on a relatively restricted display area.

4.2.3 Failure & Recovery

Generally TIP does not talk about site failures but about connection failures. For the simulation model we assume that not a complete site fails but only certain connections at that site. The connections affected are defined by the secondary of the site that fails as it is only possible to simulate secondary failures so far.
This means that the primary will time out because it did not receive a message from the secondary (although timeouts are simulated by sending a *timeout* control message).

The user can specify that the sites should always fail and always recover. One of the limitations of the simulation software so far is that the secondary at a site will only check if it should fail whenever it is about to receive a message. Future versions should allow a more flexible way of defining failures.
The decision whether to fail or not is made by comparing the actual state the secondary is in with the states that allow a failure (these states are predefined in the software, e.g. PREPARED,BEGUN). If the secondary may fail in its actual state, it will send a *timeout* message to the primary instead of the intended reply. The primary will execute the operations defined by the protocol (i.e. reach a decision) and it will send a *recover* control message to the secondary to allow it to recover. The secondary then proceeds, executing the recovery actions defined (see appendix B.2).

The difference to the same failure scenario in a traditional distributed commit protocol is that if a site fails e.g. in the ENLISTED-state, the transaction will simply be aborted and not proceed to the next state. It is therefore important that the different states that allow failures may be defined dynamically. A future version of the software must provide a user interface so that these definitions can be changed interactively.

Another important feature for a failure scenario is the possibility to define specific secondaries to fail. With TIP a communication tree is built dynamically and one site does not know about all other sites that are involved. Nevertheless it will be influenced by the failure at one site even if it did not know about its existence. The behaviour of the protocol might be quite different depending on how many secondaries fail and where they are situated in the communication tree. So far there is no user interface for this feature either but the simulation software can generally handle it.

The implementation of the recovery of a secondary does not fully reflect the possibilities that TIP offers. TIP allows the primary as well as the secondary to contact the other site and try to re-establish the connection. In the simulation, it will always be the secondary that sends a *QUERY* command which will then cause the reconnection to happen. This is not mandatory for TIP, the primary could send a *RECONNECT* without having received a *QUERY* command.

4.2.4 Problems encountered

As already mentioned above, TIP has quite a different structure to traditional distributed commit protocols. This lead to a number of problems during the implementation.

One of the problems was related to the fact that there is no notion of a coordinator and participant in TIP. TIP defines a primary and a secondary, sometimes the documentation also talks about superior and subordinate. The difficulty was to work out how they all relate to each other, opposed to the well understood relationship between a coordinator and the participants. A site may have more than 1 connection and each site controls the decision whether to abort or commit for its connections (i.e. the site acts like the coordinator, not the primary or secondary !)

Another problem is that the primary and secondary swap their roles when they use *PULL* to start a transaction. Standard 2 or 3PC protocols only allow a participant to become coordinator if the coordinator fails.

For the simulation of the traditional distributed commit protocols, the communication topology is defined before the simulation is started. This is different for TIP as well. The topology is defined by the selection of *PUSH*es, *PULL*s and*BEGIN*s and the simulation has to be able to handle this dynamic setup. It must be able to connect different sites, allowing message exchanges between all possible combinations of sites. Furthermore it must be able to handle more than one connection from one site, i.e. one site communicates with a number of other sites. This complexity caused a different composition of a site (see 4.2.2) and increased the difficulty of coordinating the processes. Despite the technical problems of how the connections can be set up and how a unique identification for the message passing can be provided, logical questions like 'What happens at connection A if connection B receives a *PREPARED* command ?' arose. Unluckily the TIP-documentation did not provide any support to solve these questions.

The different possible topologies also meant a complicated layout of the animation. The layout should be able to display e.g. a star topology as well as chains.

4.2.5 Statistics

The statistics produced by this simulation are similar to the original statistics. Some information that is irrelevant for TIP has been omitted (e.g. which site hosts the coordinator); additionally the number of message hops is displayed (see chapter 5).

Chapter 5

Simulation Scenarios

To obtain more information about the protocol, special simulation scenarios were defined and evaluated.
The scenarios should give a more detailed insight into the behaviour of the protocol than the pure analysis of the documentation.

The main problem of TIP is that it is a blocking protocol, i.e. sites may be blocked until another site recovers. Therefore scenarios in which sites fail are the most interesting scenarios to analyse. Nevertheless the performance of the protocol under normal circumstances has also been evaluated and compared to a standard 2PC protocol.

The performance of the protocol is measured by the number of messages that have been sent, the number of message rounds needed and the number of message hops.
The total number of messages sent gives a general idea about the total network traffic. The more messages are sent, the greater the burden on the network and therefore a higher possibility of congestion. And if congestion occurs, message delay is increasing.

The second unit of measurement is the number of message rounds needed. A message round is defined by the number of message pairs exchanged (e.g. $PREPARE/PREPARED = 1$ message round). With TIP, these message pairs are counted for each connection. For the total number of message rounds, the largest number of message pairs exchanged at a site will be taken and added to the results of the other sites. This gives an indication about the performance difference in the different topologies that have been evaluated. Different topologies might use the same number of messages to complete, but the message rounds are different. The more message rounds are needed, the longer it will take to complete the protocol.

In this project, we model a message round as a multicast, i.e. the same message has to be sent to a number of nodes. The communication cost and the delays for such a multicast are lower than the total cost and delay of sending the component messages individually. In this simulation, the usage of this unit of measurement has been greatly simplified. Even messages that are effectively only sent to one node will be treated like a multicast.

As a general result, chains will need more message rounds than star-topologies, even if the same number of nodes are involved and the number of messages exchanged is the same. In a star-topology, one node sends the messages to a number of other nodes, i.e. it only needs one message round to reach all nodes. In a chain topology, each node has to forward the message to the next node, which means that one message round is needed at each node. This reflects the fact that concurrency opportunities do not occur as often in a chain-topology as they do in the star-topology where a batch of messages may be sent at (roughly) the same time.

Message hops are counted as a third unit of measurement. A message hop is basically the same as a message round, only that in case of failures an additional delay is counted. If a site fails, it takes some time to recover and to reconnect to the other site. This causes a delay not only at the failed site but also at the subsequent connections as they have to wait for the preceding connection to recover. Message hops do not supply any important information about traditional 2PC because a central coordinator contacts all sites and there are no messages forwarded from one site to another, except as part of a recovery procedure (but such a recovery procedure is not realised in the simulation). Therefore message hops are only counted for TIP. In this simulation the penalty for each failure has been set to 2 additional hops (reconnect takes two message rounds).

To show the effect of different topologies, four different possibilities have been analysed (see fig. 5.2,5.3,5.4,5.5). For a basic reference, the performance of a traditional 2PC has been evaluated as well (fig. 5.1).

All of these scenarios use 10 different sites, connected in different ways. The performance is measured for cases when all sites commit, certain sites abort or sites fail. For the TIP scenarios it is possible to specify in which state a site should fail. I have restricted the scenarios to fail in the PREPARED- or the *BEGUN*-state, i.e. the secondary will fail before it receives the *COMMIT* or *ABORT* message from the primary. A failure in the *BEGUN*-state and any other state would simply lead to abort the transaction without involving a special recovery procedure. Only if the secondary fails in the *PREPARED*-state it has to contact the primary again to find out the final decision. This additional communication will influence the performance not only at the directly concerned site but also at subsequent connections.

A number of different simulations have been defined, listed below. Standard 2PC has only been evaluated for some of them (1.1, 1.2, 1.3, 1.14, 1.15) because voting to abort at different sites does not influence the behaviour of the protocol as a central coordinator exists. Other simulations could not be run because it is not possible to define failures at specific sites for the standard 2PC simulation.

The chosen simulations are as follows:

1. no failure, commit
2. no failure, abort site 1
3. no failure, abort site 3
4. no failure, abort site 5
5. no failure, abort site 8
6. no failure, abort site 10
7. failure site 3, commit
8. failure site 3, abort site 2
9. failure site 5, commit
10. failure site 3 and 8, commit
11. failure site 8, commit
12. failure site 8, abort at site 6
13. failure site 10, commit
14. always fail and always recover, commit
15. always fail and always recover, abort site 3

For simulations 14 and 15, 'always fail and always recover' generally means that each site fails in every state. This would mean that in a TIP scenario all sites would already fail in the ENLISTED-state which in turn leads to abort of the transaction. To allow the protocol to continue with the 2PC, failures have been restricted to occur only in the PREPARED-state.

5.1 Evaluated Scenarios and Results

Below the different scenarios that have been simulated are described. The results are shown in tables 5.1/5.2/5.3.

Scenario 1:

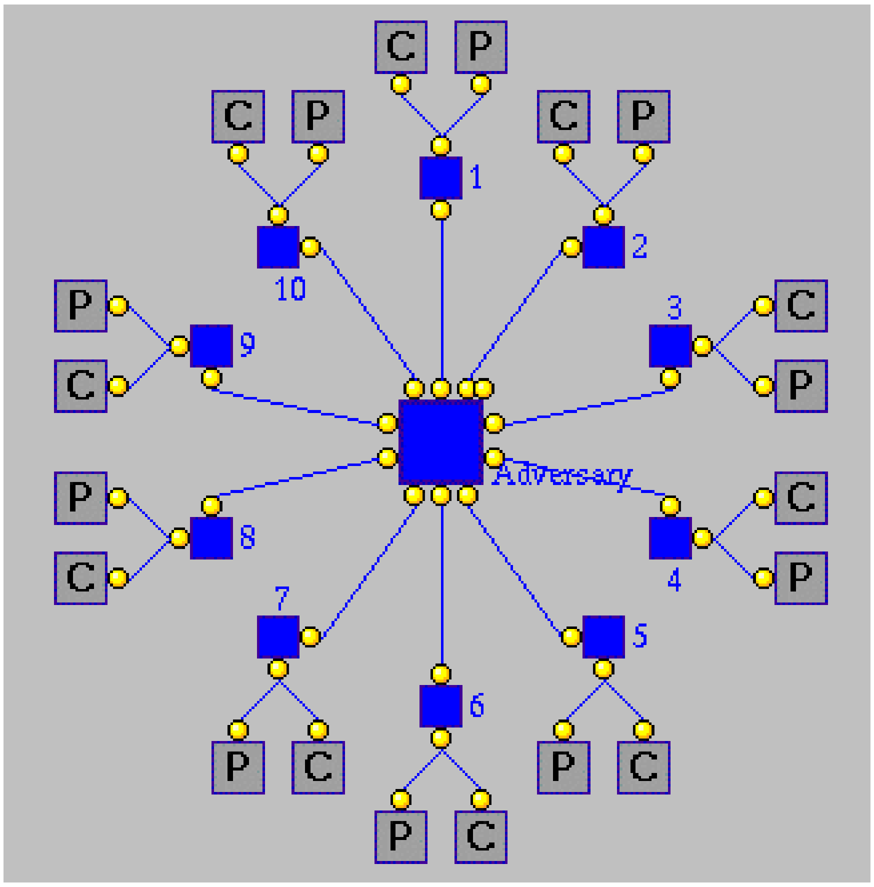


Figure 5.1: Scenario 1 - Traditional 2PC

This scenario describes the performance of a standard 2PC protocol. It uses ten participants, connected as shown in fig. 5.1.

Simulation 1.1: no failure, commit

Simulation 1.2: no failure, abort site 1

With traditional 2PC, the results for an abort decision at different sites will always be the same (see 1.3).

This simulation may therefore be used for comparison with TIP simulations that abort at different sites

Simulation 1.3: no failure, abort site 3

This simulation gives the same result as 1.2.

Simulation 1.14: always fail and always recover, commit

Simulation 1.15: always fail and always recover, abort site 3

Scenario 2:

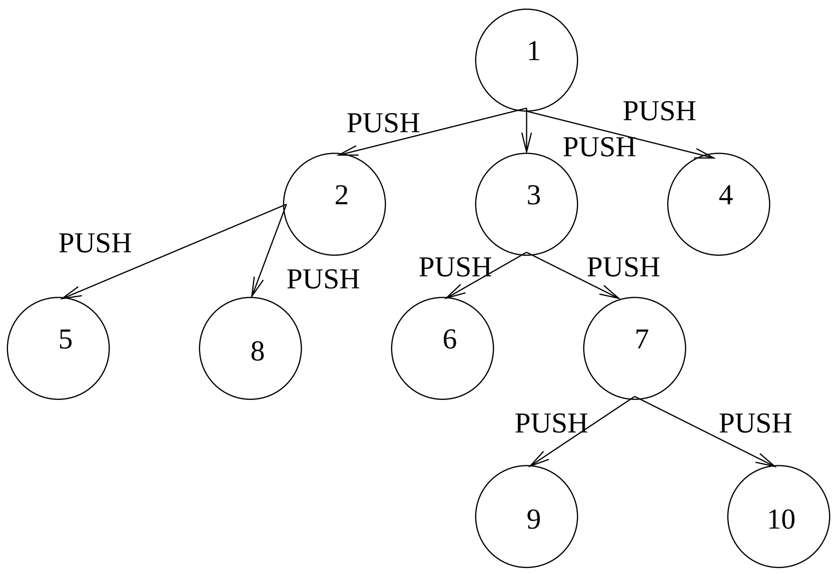


Figure 5.2: Scenario 2 - topology

This scenario describes the first of the four different TIP topologies that have been analysed. As for the standard 2PC, the communication tree is built of ten sites, using following commands:

1 *PUSH* 2,3 and 4

2 *PUSH* 5 and 8

3 *PUSH* 6 and 7

7 *PUSH* 9 and 10

Simulation 2.1: no failure, commit

Simulation 2.2: no failure, abort site 1

Simulation 2.3: no failure, abort site 3

Secondary at site 3 waits until all subsequent branches completed 'Abort' before itself aborts. This is one of the cases where the lack of TIP documentation becomes obvious - the implementation could be changed so that site 3 aborts without waiting for results from subsequent connections.

Simulation 2.4: no failure, abort site 5

Simulation 2.5: no failure, abort site 8

Simulation 2.6: no failure, abort site 10

Simulation 2.7: failure site 3, commit

Simulation 2.8: failure site 3, abort site 2

Simulation 2.9: failure site 5, commit

The result of this simulation is the same as for 2.11 (failure at site 8) because both sites are final nodes at the same depth of the tree

Simulation 2.10: failure site 3 and 8, commit

Simulation 2.11: failure site 8, commit

Secondary at site 2 waits until the primary of connection with site 8 has recovered and completed the transaction - this is another example for implementation-depending behaviour.

Simulation 2.12: failure site 8, abort site 6

Simulation 2.13: failure site 10, commit

Simulation 2.14: always fail and always recover (all secondaries in PREPARED-state), commit

Simulation 2.15: always fail and always recover (all sites), abort site 3

The branches that are connected with site 3 finish earlier as the other branches because they are connected to the first sites that fail (6,7).

Site 1 has to wait for a result from site 3. Due to the abort decision, subsequent sites of site 3 do not wait for the confirmation from site 1 (site 3 has to reply *ABORTED* to *PREPARE* command)

Scenario 3:

This scenario consists of 3 sub-scenarios to demonstrate the influence of different commands to generate the communication tree. The topology itself remains the same, just that in one scenario a 1PC is used to start the transaction, another scenario uses *PULL* and the third one *PUSH*es all transactions.

Sub-Scenario 3.1:

In this scenario, the communication tree has been built using a 1PC for the first connection:

1 *BEGIN* 2

2 *PUSH* 3

3 *PUSH* 4,5 and 6

5 *PUSH* 7,8 and 9

8 *PUSH* 10

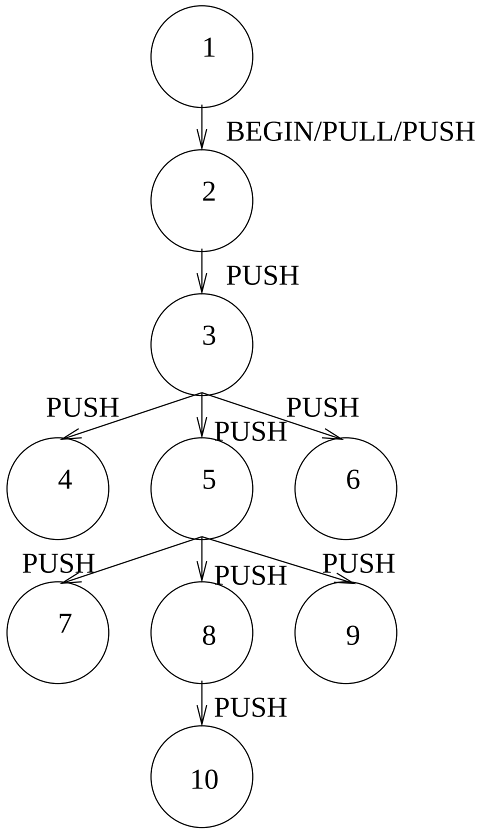


Figure 5.3: Scenario 3 - topology

Simulation 3.1.1: no failure, commit

Simulation 3.1.2: no failure, abort at site 1

Site 1 begins transaction with site 2 and aborts without waiting for decisions from the other sites.

As the overall result has to be abort this is not a real problem,

just again a behaviour that is not clearly defined in the protocol description.

Simulation 3.1.3: no failure, abort site 3

Simulation 3.1.4: no failure, abort site 5

Simulation 3.1.5: no failure, abort site 8

Simulation 3.1.6: no failure, abort site 10

Simulation 3.1.7: failure site 3, commit

Simulation 3.1.8: failure site 3, abort site 2

Simulation 3.1.9: failure site 5, commit

Simulation 3.1.10: failure site 3 and 8, commit

Simulation 3.1.11: failure site 8, commit

Simulation 3.1.12: failure site 8, abort site 6

Simulation 3.1.13: failure site 10, commit

Simulation 3.1.14: always fail and always recover (first secondary fails in BEGUN-state, all other secondaries in PREPARED-state), commit
First connection is already back in IDLE-state after aborting whilst subsequent connections are still *PUSH*ing and *PULL*ing
Recovery: *QUERY* - *QUERIEDNOTFOUND* leads to global abort because of failure in BEGUN-state for first connection

Simulation 3.1.15: always fail and always recover (all sites), abort site 3

Sub-Scenario 3.2:

The second scenario for this topology uses *PULL* to set up the first connection.

topology:
1 *PULL* 2
2 *PUSH* 3
3 *PUSH* 4,5 and 6
5 *PUSH* 7,8 and 9
8 *PUSH* 10

As already mentioned several times, there are a number of issues missing in the available TIP documentation when it comes to the actual implementation. In this case it is related to the way *PUSH* and *PULL* branches are linked. In this simulation, *COMMIT* goes down the *PUSH* branches before *PULL* branch is informed and proceeds (i.e. primary of the *PUSH* branch is already IDLE, whereas the primary of the *PULL* branch is still in PREPARED-state). This could be handled more efficiently if both connections would send the *COMMIT* command concurrently.

Simulation 3.2.1: no failure, commit

Simulation 3.2.2: no failure, abort site 1

Simulation 3.2.3: no failure, abort site 3

Simulation 3.2.4: no failure, abort site 5

Simulation 3.2.5: no failure, abort site 8

Simulation 3.2.6: no failure, abort site 10

Simulation 3.2.7: failure site 3, commit

Simulation 3.2.8: failure site 3, abort site 2

Simulation 3.2.9: failure site 5, commit

Simulation 3.2.10: failure site 3 and 8, commit

Simulation 3.2.11: failure site 8, commit

Simulation 3.2.12: failure site 8, abort site 6

Simulation 3.2.13: failure site 10, commit

Simulation 3.2.14: always fail and always recover (all secondaries in PREPARED-state), commit

Simulation 3.2.15: always fail and always recover (all sites), abort site 3
NOTE: failures for *PULL*ed connections have not been realised. This means that neither for this simulation nor for simulation 3.2.14 a failure at site 1 or site 2 occurs.
Consequently, no recovery and reconnection messages are exchanged.

Sub-Scenario 3.3:

The third possibility to generate the transaction tree is by using only *PUSH*es.

topology:

1 *PUSH* 2

2 *PUSH* 3

3 *PUSH* 4,5 and 6

5 *PUSH* 7,8 and 9

8 *PUSH* 10

Simulation 3.3.1: no failure, commit

Simulation 3.3.2: no failure, abort site 1

Simulation 3.3.3: no failure, abort site 3

Simulation 3.3.4: no failure, abort site 5

Simulation 3.3.5: no failure, abort site 8

Simulation 3.3.6: no failure, abort site 10

Simulation 3.3.7: failure site 3, commit

Simulation 3.3.8: failure site 3, abort site 2

Simulation 3.3.9: failure site 5, commit

Simulation 3.3.10: failure site 3 and 8, commit

Simulation 3.3.11: failure site 8, commit

Simulation 3.3.12: failure site 8, abort site 6

Simulation 3.3.13: failure site 10, commit

Simulation 3.3.14: always fail and always recover (all secondaries in PREPARED-state), commit

Simulation 3.3.15: always fail and always recover (all sites), abort site 3

Scenario 4:

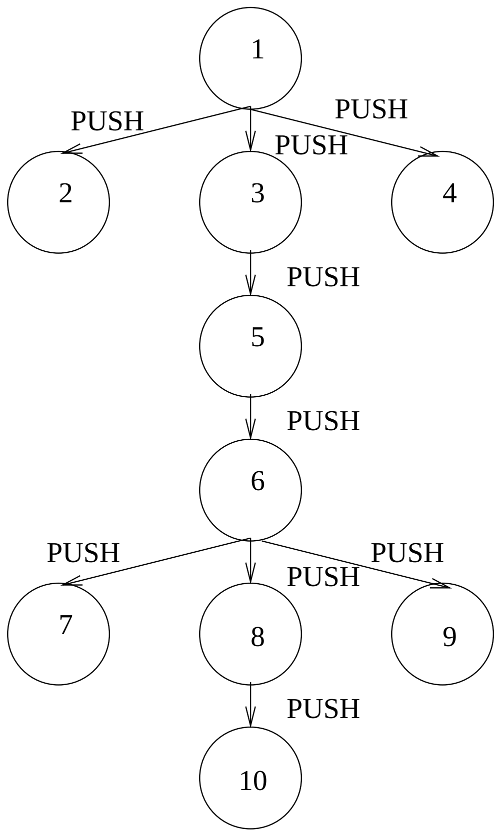


Figure 5.4: Scenario 4 - topology

topology:

1 *PUSH* 2,3 and 4

3 *PUSH* 5

5 *PUSH* 6

6 *PUSH* 7,8 and 9

8 *PUSH* 10

Simulation 4.1: no failure, commit

Simulation 4.2: no failure, abort site 1

Simulation 4.3: no failure, abort site 3

Simulation 4.4: no failure, abort site 5

Simulation 4.5: no failure, abort site 8

Simulation 4.6: no failure, abort site 10

Simulation 4.7: failure site 3, commit

Simulation 4.8: failure site 3, abort site 2

Simulation 4.9: failure site 5, commit

Simulation 4.10: failure site 3 and 8, commit

Simulation 4.11: failure site 8, commit

Simulation 4.12: failure site 8, abort site 6

Simulation 4.13: failure site 10, commit

Simulation 4.14: always fail and always recover (all secondaries
in PREPARED-state), commit

Simulation 4.15: always fail and always recover (all sites), abort site 3

Scenario 5:

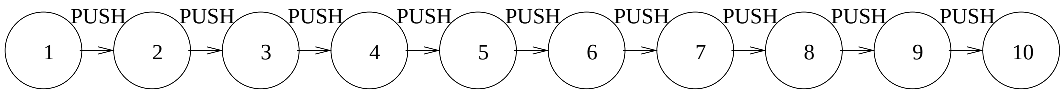


Figure 5.5: Chain topology

topology:

1 *PUSH* 2

2 *PUSH* 3

3 *PUSH* 4

4 *PUSH* 5

5 *PUSH* 6

6 *PUSH* 7

7 *PUSH* 8

8 *PUSH* 9

9 *PUSH* 10

Simulation 5.1: no failure, commit

Simulation 5.2: no failure, abort site 1

Simulation 5.3: no failure, abort site 3

Simulation 5.4: no failure, abort site 5

Simulation 5.5: no failure, abort site 8

Simulation 5.6: no failure, abort site 10

Simulation 5.7: failure site 3, commit

Simulation 5.8: failure site 3, abort site 2

Simulation 5.9: failure site 5, commit

Simulation 5.10: failure site 3 and 8, commit

Simulation 5.11: failure site 8, commit

Simulation 5.12: failure site 8, abort site 6

Simulation 5.13: failure site 10, commit

Simulation 5.14: always fail and always recover (all secondaries
in PREPARED-state), commit

Simulation 5.15: always fail and always recover (all sites), abort site 3

5.2 Discussion of Results

To be able to compare the TIP results with the standard 2PC, some adjustments have to be done to the figures:
TIP starts with commands to establish the connection. These commands are *IDENTIFY*, *IDENTIFIED*, *PUSH*, *PUSHED*, *PULL*, *PULLED*, *BEGIN*, *BEGUN*. These sort of commands are not known with the standard 2PC simulation and have therefore to be reduced from the total number of messages sent and also from the number of message rounds. The simulation does not use *IDENTIFY* and *IDENTIFIED*, therefore only 2 messages per connection have to be reduced from the total number of messages sent. Each TIP scenario consists of 9 connections, therefore 18 (9*2) messages have to be deducted.
The number of message rounds is calculated using the number of rounds of one connection at each site. This means that one message round has to be subtracted for each site that hosts a primary, i.e. 4 rounds for scenario 2, 5 rounds for scenarios 3 and 4 and 9 rounds for scenario 5.

Message rounds are defined differently in standard 2PC and TIP. With standard 2PC, a central coordinator sends and receives messages and each exchange is a message round. For TIP the message rounds are counted per connection at each site. These rounds are then compared and the largest one will be taken. Finally, the results of all sites are added up.

The failure scenario 'always fail and always recover' with standard 2PC means that a site fails, recovers and proceeds with the protocol, whereas TIP is a presumed-abort protocol and a reconnection is only needed if the secondary did not receive the final decision from the primary, i.e. the two sites will only reconnect if the connection failed in the PREPARED state.

With these specialities in mind, a number of observations can be made. Probably the most obvious one is that in our simulation model standard 2PC has a constant number of message rounds (4) under all circumstances whereas for TIP this depends on the topology as well as the (failure) scenario.

Generally, standard 2PC as simulated in this simulation software package always needs four message rounds to complete and the number of messages sent in a scenario with no failures and a final decision to commit can be calculated as:
number of message rounds * number of participants.
The number of messages sent is reduced by two for each participant that votes to abort because these participants do not have to be informed about the final decision and do not have to confirm it.
TIP needs three message rounds per connection to complete a 2PC if no failures occur. The total number of message rounds is defined by
number of message rounds per connection * number of sites that initiated at least one connection (i.e. that host at least one primary) if no site votes to abort. In that case the number of messages sent is defined as:
2 * number of message rounds (3) * number of connections
(two messages are exchanged per message round).[1]

If the final decision is to abort, the TIP results depend much more on the topology.

The chain-topology simulated with TIP shows a good performance concerning the number of messages exchanged. In some of the simulations (4,5,6 and 12) it even has the lowest number of messages sent. But on the other hand it always needs the most message rounds to complete. The large number of message rounds needed is due to the fact that messages are forwarded from one node to another and not centrally from one node to all others (or sub-groups; therefore no multicast is possible). Concerning the number of messages sent, they decrease the deeper down in the tree a site decides to abort. If one site decides to abort, all precedent connections can already abort in the ENLISTED-state and therefore less messages are needed. For the chain topology this in turn leads to less message rounds as well because there exists only one outgoing connection at each site.

The number of messages sent decreases for all topologies if the final decision is to abort (except if the root aborts as it only hosts primaries and they will execute the complete 2PC regardless of the final decision) but the total number of message rounds does only change in the chain topology. As the largest number of message rounds is taken into account for each site, the smaller number of rounds needed in the case of an abort are not noticeable. This would change if all connections from one site decide to abort.

For standard 2PC it does not make any difference which site votes to abort, the total number

[1] For topologies that include connections employing a 1PC the definitions will of course be different (see scenario 3.1.1) !

of messages sent is always the same. For TIP different number of messages may be sent, the differences depend on the topology chosen (see above).

The number of message hops become only important in case of failures, in any other case they will always be the same as the number of message rounds.

With TIP, failures at different sites do not influence the number of messages sent or message rounds because these units of measurement do not reflect the influence on the delay, i.e. if one site fails, other sites are blocked. Depending on how long the failed site needs to recover and how many other sites are influenced, the overall performance will decrease. The number of message hops used shows that the delay increases with the number of subsequent sites after a site that failed. Also, the more sites fail, the higher the delay will be because all delays add up (see simulations 10 and 14).

If all sites fail (and recover), an abort at one site reduces the results significantly (except for sub-scenario 3.1 because the employed 1PC leads to abort in both cases). The reason is that the site that votes 'abort' does not fail and sites at a higher level in the tree do not have to reconnect. They simply exchange *QUERY* and *QUERIEDNOTFOUND* messages which inform the secondary that it has to abort (no *RECONNECT/RECONNECTED* necessary).

Table 5.1: Simulation results

Scenario	Intended decision	Real decision	Messages sent	Message rounds	Message hops
No Failures, all sites vote 'commit'					
1.1	Commit	Commit	40	4	-
2.1	Commit	Commit	54-18 = 36	12-4 = 8	12
3.1.1	Commit	Commit	52-18 = 34	14-5 = 9	14
3.2.1	Commit	Commit	54-18 = 36	15-5 = 10	15
3.3.1	Commit	Commit	54-18 = 36	15-5 = 10	15
4.1	Commit	Commit	54-18 = 36	15-5 = 10	15
5.1	Commit	Commit	54-18 = 36	27-9 = 18	27
No failure, site 1 votes 'abort'					
1.2	Abort	Abort	38	4	-
2.2	Abort	Abort	54-18 = 36	12-4 = 8	12
3.1.2	Abort	Abort	52-18 = 34	14-5 = 9	14
3.2.2	Abort	Abort	52-18 = 34	14-5 = 9	14
3.3.2	Abort	Abort	54-18 = 36	15-5 = 10	15
4.2	Abort	Abort	54-18 = 36	15-5 = 10	15
5.2	Abort	Abort	54-18 = 36	27-9 = 18	27
No Failures, site 3 votes 'abort'					
1.3	Abort	Abort	38	4	-
2.3	Abort	Abort	52-18 = 34	12-4 = 8	12
3.1.3	Abort	Abort	50-18 = 32	13-5 = 8	13
3.2.3	Abort	Abort	52-18 = 34	14-5 = 9	14
3.3.3	Abort	Abort	50-18 = 32	13-5 = 8	13
4.3	Abort	Abort	52-18 = 34	15-5 = 10	15
5.3	Abort	Abort	50-18 = 32	25-9 = 16	25
No failure, site 5 votes 'abort'					
2.4	Abort	Abort	50-18 = 32	12-4 = 8	12
3.1.4	Abort	Abort	48-18 = 30	13-5 = 8	13
3.2.4	Abort	Abort	50-18 = 32	14-5 = 9	14
3.3.4	Abort	Abort	48-18 = 30	13-5 = 8	13
4.4	Abort	Abort	50-18 = 32	14-5 = 9	14
5.4	Abort	Abort	46-18 = 28	23-9 = 14	23
No failure, site 8 votes 'abort'					
2.5	Abort	Abort	50-18 = 32	12-4 = 8	12
3.1.5	Abort	Abort	46-18 = 28	13-5 = 8	13
3.2.5	Abort	Abort	48-18 = 30	14-5 = 9	14
3.3.5	Abort	Abort	46-18 = 28	13-5 = 8	13
4.5	Abort	Abort	46-18 = 28	13-5 = 8	13
5.5	Abort	Abort	40-18 = 22	20-9 = 11	20
No failure, site 10 votes 'abort'					
2.6	Abort	Abort	48-18 = 30	12-4 = 8	12
3.1.6	Abort	Abort	44-18 = 26	12-5 = 7	12
3.2.6	Abort	Abort	46-18 = 28	13-5 = 8	13
3.3.6	Abort	Abort	44-18 = 26	12-5 = 7	12
4.6	Abort	Abort	44-18 = 26	12-5 = 7	12
5.6	Abort	Abort	36-18 = 18	18-9 = 9	18

Table 5.2: Simulation results

Scenario	Intended decision	Real decision	Messages sent	Message rounds	Message hops
Failure at site 3, all sites vote 'commit'					
2.7	Commit	Commit	59-18 = 41	14-4 = 10	18
3.1.7	Commit	Commit	57-18 = 39	16-5 = 11	22
3.2.7	Commit	Commit	59-18 = 41	17-5 = 12	23
3.3.7	Commit	Commit	59-18 = 41	17-5 = 12	23
4.7	Commit	Commit	59-18 = 41	17-5 = 12	25
5.7	Commit	Commit	59-18 = 41	29-9 = 20	43
Failure at site 3, site 2 votes 'abort'					
2.8	Abort	Abort	53-18 = 35	12-4 = 8	16
3.1.8	Abort	Abort	53-18 = 35	14-5 = 9	20
3.2.8	Abort	Abort	55-18 = 37	15-5 = 10	21
3.3.8	Abort	Abort	53-18 = 35	14-5 = 9	20
4.8	Abort	Abort	53-18 = 35	15-5 = 10	23
5.8	Abort	Abort	53-18 = 35	26-9 = 17	40
Failure at site 5, all sites vote 'commit'					
2.9	Commit	Commit	59-18 = 41	14-4 = 10	14
3.1.9	Commit	Commit	57-18 = 39	16-5 = 11	20
3.2.9	Commit	Commit	59-18 = 41	17-5 = 12	21
3.3.9	Commit	Commit	59-18 = 41	17-5 = 12	21
4.9	Commit	Commit	59-18 = 41	17-5 = 12	23
5.9	Commit	Commit	59-18 = 41	29-9 = 20	39
Failure at site 3 and 8, all sites vote 'commit'					
2.10	Commit	Commit	64-18 = 46	16-4 = 12	20
3.1.10	Commit	Commit	62-18 = 44	18-5 = 13	26
3.2.10	Commit	Commit	64-18 = 46	19-5 = 14	27
3.3.10	Commit	Commit	64-18 = 46	19-5 = 14	27
4.10	Commit	Commit	64-18 = 46	19-5 = 14	29
5.10	Commit	Commit	64-18 = 46	31-9 = 22	49
Failure at site 8, all sites vote 'commit'					
2.11	Commit	Commit	59-18 = 41	14-4 = 10	14
3.1.11	Commit	Commit	57-18 = 39	16-5 = 11	18
3.2.11	Commit	Commit	59-18 = 41	17-5 = 12	19
3.3.11	Commit	Commit	59-18 = 41	17-5 = 12	19
4.11	Commit	Commit	59-18 = 41	17-5 = 12	19
5.11	Commit	Commit	59-18 = 41	29-9 = 20	33
Failure at site 8, site 6 votes 'abort'					
2.12	Abort	Abort	51-18 = 33	12-4 = 8	12
3.1.12	Abort	Abort	49-18 = 31	13-5 = 8	15
3.2.12	Abort	Abort	51-18 = 33	14-5 = 9	16
3.3.12	Abort	Abort	49-18 = 31	13-5 = 8	15
4.12	Abort	Abort	49-18 = 31	13-5 = 8	15
5.12	Abort	Abort	45-18 = 27	22-9 = 13	26

Table 5.3: Simulation results

Scenario	Intended decision	Real decision	Messages sent	Message rounds	Message hops
Failure at site 10, all sites vote 'commit'					
2.13	Commit	Commit	59-18 = 41	14-4 = 10	14
3.1.13	Commit	Commit	57-18 = 39	16-5 = 11	16
3.2.13	Commit	Commit	59-18 = 41	17-5 = 12	17
3.3.13	Commit	Commit	59-18 = 41	17-5 = 12	17
4.13	Commit	Commit	59-18 = 41	17-5 = 12	17
5.13	Commit	Commit	59-18 = 41	29-9 = 20	29
Always fail and always recover, all sites vote 'commit'					
1.14	Commit	Abort	79	4	-
2.14	Commit	Commit	99-18 = 81	20-4 = 16	28
3.1.14	Commit	Abort	59-18 = 41	13-5 = 8	33
3.2.14	Commit	Commit	94-18 = 76	23-5 = 18	35
3.3.14	Commit	Commit	99-18 = 81	25-5 = 20	45
4.14	Commit	Commit	99-18 = 81	25-5 = 20	45
5.14	Commit	Commit	99-18 = 81	45-9 = 36	117
Always fail and always recover, site 3 votes 'abort'					
1.15	Abort	Abort	79	4	-
2.15	Abort	Abort	60-18 = 42	12-4 = 8	16
3.1.15	Abort	Abort	56-18 = 38	12-5 = 7	20
3.2.15	Abort	Abort	59-18 = 41	14-5 = 9	20
3.3.15	Abort	Abort	57-18 = 39	13-5 = 8	19
4.15	Abort	Abort	60-18 = 42	15-5 = 10	27
5.15	Abort	Abort	57-18 = 39	25-9 = 16	67

Chapter 6

Conclusions

The approach taken by the TIP group to deliver a distributed commit protocol suitable for usage in the Internet defines a different way to control distributed transactions. It achieves its simplicity by separating the main communication issues from the distributed commit protocol itself. This procedure is reasonable as it allows to concentrate on the main issues of a distributed commit protocol without being 'disturbed' by problems that do not relate directly to the protocol. Nevertheless there are a number of issues that are overlapping and should be addressed in the definition of the distributed commit protocol. These issues are mainly related to the duties of the application-to-application communication and security concerns. In the documentation of TIP it is very often assumed that problems are solved by other means than the protocol itself without describing the impact on the protocol (e.g. consequences if not all requested servers joined the transaction before the commit is being issued).

Despite the fact that issues of this sort have been delayed for later solution, there are still a number of other problems with TIP.

The drawbacks of TIP mainly stem from the flexibility in which the communication tree is built. The applications are responsible for contacting the servers and it is therefore possible to have the same transaction (i.e. the same data items are updated) executed in completely different communication topologies. As the topology influences e.g. the message delay (messages have to be forwarded down to the last site and answers have to 'travel' all the way back - therefore the longer the chain the slower the response time) this determines the overall performance as well.

Another interesting factor when studying TIP is the behaviour in case of failures. TIP does not give the possibility for a server to contact any other server that is included in the same transaction. All communication is strict point-to-point and no information about which servers are included is distributed. That means that TIP is very vulnerable to single-point-failures. If one site fails after the subordinates have voted to commit but before sending the final decision, all connections will be blocked. They return to the IDLE-state and could use the connection for other transactions but e.g. locks on data cannot be released. To complete the transaction, the two servers have to get in contact again and find out about the final outcome of the transaction.

The blocking behaviour of TIP is a big disadvantage especially in an internet environment because the reliability of other servers is difficult to assess and no enterprise wants its system to be blocked because of the failure in another enterprise's system. The lack of any intention to overcome this problem is a step backwards compared to the traditional commit protocols.

On the other hand one must be aware that termination protocols as they are known from traditional distributed commit protocols are also not directly transferrable to the Internet usage. A central coordinator can lead to other problems (e.g. congestion) that might outweigh the disadvantage of a blocked system. The importance of special termination protocols and avoidance of a blocked system depends on the reliability of the servers that are involved in a transaction.

TIP is still under development and the developers are aware that a number of open questions need to be addressed. A lot of work is still to be invested into the proper definition of the protocol, the requirements and possible scenarios in which it could be used. The documentation and information provided so far gives a good starting point for discussions and implementations but needs to be refined.

It remains to be seen if the restricted functionality that TIP is designed to offer will find the acceptance it is aiming for. This will depend very much on the environment it will be used in. Although it is designed for usage in the Internet the blocking behaviour might detract from its suitability. The environment it will be used in must still be fairly predictable to guarantee an efficient performance and allow security issues to be addressed (e.g. restrict access to 'trusted' systems). This might underline the suitability of the protocol for companies that use distributed transaction processing only for simple cases and cannot afford the investment needed for more sophisticated solutions. On the other hand, it is not clear how TIP will actually be implemented. It is one of the requirements for TIP that it should be simple and easily implemented, therefore it does not seem to be suitable to include it in a TPM but the documentation does not explain what form TIP will take and in which part of the system it is supposed to be implemented.

Bibliography

[1] J. Lyon, K. Evans, J. Klein: *Transaction Internet Protocol*
(ftp://ds.internic.net/internet-drafts/draft-lyon-itp-nodes-01.txt
replaced by draft-lyon-itp-nodes.02.txt)

[2] J. Lyon, K. Evans, J. Klein: *Transaction Internet Protocol - Requirements*
(ftp://ds.internic.net/internet-drafts/draft-evans-tip-functions-00.txt
replaced by draft-evans-tip-functions-01.txt)

[3] K. Evans, J. Klein, J. Lyon, F. Upton: *TIP: Gateway to Heterogeneous Internet Transactions*

[4] S. Spiro: *SCP - Session Control Protocol V 1.1* (http://sunsite.unc.edu/ses/scp.html)

[5] A. Freier, P. Karlton, P. Kocher: Internet Draft: *The SSL Protocol Version 3.0*

[6] A. Freier, P. Karlton, P. Kocher, T. Dierks: Internet Draft: *The TLS protocol*
(ftp://ds.internic.net/internet-drafts/draft-ietf-tls-protocol-03.txt)

[7] P. Bernstein, E. Newcomer: *Principles of Transaction Processing*
Morgan-Kaufmann, 1997

[8] P. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems*

[9] META Group Consulting: *Electronic Commerce in the Financial Transactions Processing Market*, October 1996

[10] J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*
Morgan Kaufmann, 1993

[11] Open Systems Interconnection - Recommendation—International Standard: ISO/IEC 9805-1:1997 (E), International Standard, ITU-T rec. X.852 (1997 E), ITU-T Recommendation

[12] R. Orfali, D. Harkey, J. Edwards: *The Essential Distributed Objects Survival Guide*

[13] R. McNab, F. Howell: *Using Java for Discrete Event Simulation*
in proc. Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh, 219-228
(http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.0)

[14] S. Ramsay: *Simulation of Distributed Commit Protocols*
University of Edinburgh, May 1997

[15] R. Milner: *Communication and Concurrency*
Prentice-Hall, Hertfordshire, 1989

Appendix A

Abbreviations

CORBA	- Common Object Request Broker Architecture
GUI	- Graphical User Interface
HTTP	- HyperText Transfer Protocol
id	- identifier
IETF	- Internet Engineering Task Force
OSI	- Open Systems Interconnection
OTS	- Object Transaction Service
PC	- Phase Commit
TCP	- Transfer Control Protocol
TIP	- Transaction Internet Protocol
TM	- Transaction Manager
TMP	- TIP Multiplexing Protocol
TP	- Transaction Processing
TPM	- Transaction Processing Monitor
txn	- Transaction
URL	- Uniform Resource Locator

Appendix B

Code Listings

NOTE: The code listed below shows how the main parts of the TIP simulation have been implemented. The implementation does only cover parts of the protocol but gives some indications where extensions for a complete implementation could be introduced.

B.1 Table_TIP.java

```
/*
** Author : Tanja Schulz <tss@dcs.ed.ac.uk>, 12/06/97
** Files  : Table_TIP.java, Table_TIP.class
** Package: simdcp
** Update : 12/06/97
**
** Description: A specific protocol represented in the table structure.
** Name        : Transaction Internet Protocol
** Comments    :
*/

package eduni.simdcp;

/* Methods defined in final class Table_TIP
** PUBLIC
** Table_TIP(String)
**
** PROTECTED
**       void    Display_Stats(int)
**
** PRIVATE
**       void    SetUp_Table()
**       void    Set_State_Names()
**       void    Set_Message_Names()
**       void    Set_Log_Names()
**       void    Set_Method_Names()
**       void    Add_Actions()
*/
final class Table_TIP extends Table implements Constants {
```

```
public Table_TIP(String name) {
    super.Init_Table(name);                  // Must be first in constructor
    this.Setup_Table();
}                                                // constructor Table_TIP(1)

private void Setup_Table() {
    Info("Constructing table: "+super.Name()+"...");
    this.Set_State_Names();
    this.Set_Message_Names();
    this.Set_Log_Names();
    this.Set_Method_Names();
    super.Add_States();
    super.Set_Recover_State(PRIMARY, 13);
    super.Set_Recover_State(SECONDARY, 13);
    this.Add_Actions();
    Info("...Finished "+super.Name());
}                                                     // void Sup_Table(0)

private void Set_State_Names() {
  Set_State_Name(PRIMARY, 1, "INITIAL",      false);
  Set_State_Name(PRIMARY, 2, "IDLE",         false);   // 4th parameter =
  Set_State_Name(PRIMARY, 3, "BEGUN",        false);   // indicates failure
  Set_State_Name(PRIMARY, 4, "BEGUN",        false);   // allowed (true) -
  Set_State_Name(PRIMARY, 5, "ENLISTED",     false);   // not allowed (false)
  Set_State_Name(PRIMARY, 6, "PREPARED",     false);
  Set_State_Name(PRIMARY, 7, "ERROR",        false);
  Set_State_Name(PRIMARY, 8, "ENLISTED_S",   false);
  Set_State_Name(PRIMARY, 9, "PREPARED_S",   false);
  Set_State_Name(PRIMARY, 10, "ENLISTED",    false);
  Set_State_Name(PRIMARY, 11, "PREPARED",    false);
  Set_State_Name(PRIMARY, 12, "ENLISTED_S",  false);
  Set_State_Name(PRIMARY, 13, "RECOVER",     false);
  Set_State_Name(PRIMARY, 14, "IDLE",        false);
  Set_State_Name(PRIMARY, 15, "PREPARED",    false);
  Set_State_Name(PRIMARY, 16, "BEGUN",       false);
  Set_State_Name(PRIMARY, 17, "RECOVER",     false);

  Set_State_Name(SECONDARY, 1, "INITIAL",      false);
  Set_State_Name(SECONDARY, 2, "IDLE",         false);
  Set_State_Name(SECONDARY, 3, "BEGUN",        true );
  Set_State_Name(SECONDARY, 4, "BEGUN",        true );
  Set_State_Name(SECONDARY, 5, "ENLISTED",     false);
  Set_State_Name(SECONDARY, 6, "PREPARED",     true );
  Set_State_Name(SECONDARY, 7, "ERROR",        false);
  Set_State_Name(SECONDARY, 8, "ENLISTED_P",   false);
  Set_State_Name(SECONDARY, 9, "PREPARED_P",   false);
  Set_State_Name(SECONDARY, 10, "ENLISTED_P",  false);
  Set_State_Name(SECONDARY, 11, "PREPARED_P",  false);
  Set_State_Name(SECONDARY, 12, "ENLISTED",    false);
  Set_State_Name(SECONDARY, 13, "RECOVER",     false);
  Set_State_Name(SECONDARY, 14, "IDLE",        false);

  Set_State_Name(CONNECTION, 1, "INITIAL",      false);
```

```
  Set_State_Name(CONNECTION, 2, "IDLE",         false);
  Set_State_Name(CONNECTION, 3, "BEGUN",        false);
  Set_State_Name(CONNECTION, 4, "MULTIPLEXING", false);
  Set_State_Name(CONNECTION, 5, "ENLISTED",     false);
  Set_State_Name(CONNECTION, 6, "PREPARED",     false);
  Set_State_Name(CONNECTION, 7, "ERROR",        false);
}                                                 // void Set_State_Names(0)

private void Set_Message_Names() {
  Set_Message_Name(1,  "START");
  Set_Message_Name(2,  "IDENTIFY");
  Set_Message_Name(3,  "IDENTIFIED");
  Set_Message_Name(4,  "BEGIN");
  Set_Message_Name(5,  "BEGUN");
  Set_Message_Name(6,  "NOTBEGUN");
  Set_Message_Name(7,  "MULTIPLEX");
  Set_Message_Name(8,  "MULTIPLEXING");
  Set_Message_Name(9,  "CANTMULTIPLEX");
  Set_Message_Name(10, "PUSH");
  Set_Message_Name(11, "PUSHED");
  Set_Message_Name(12, "ALREADYPUSHED");
  Set_Message_Name(13, "NOTPUSHED");
  Set_Message_Name(14, "PULL");
  Set_Message_Name(15, "PULLED");
  Set_Message_Name(16, "NOTPULLED");
  Set_Message_Name(17, "QUERY");
  Set_Message_Name(18, "QUERIEDEXISTS");
  Set_Message_Name(19, "QUERIEDNOTFOUND");
  Set_Message_Name(20, "RECONNECT");
  Set_Message_Name(21, "RECONNECTED");
  Set_Message_Name(22, "NOTRECONNECTED");
  Set_Message_Name(23, "ABORT");
  Set_Message_Name(24, "ABORTED");
  Set_Message_Name(25, "COMMIT");
  Set_Message_Name(26, "COMMITTED");
  Set_Message_Name(27, "PREPARE");
  Set_Message_Name(28, "PREPARED");
  Set_Message_Name(29, "READONLY");
  Set_Message_Name(30, "ERROR");
}                                               // void Set_Message_Names(0)

private void Set_Log_Names() {
    Set_Log_Name(1, "Port");
    Set_Log_Name(2, "Maxtrans");
    Set_Log_Name(3, "Forget");
    Set_Log_Name(4, "Commit");
    Set_Log_Name(5, "Prepare");
}                                                   // void Set_Log_Names(0)

private void Set_Method_Names() {
  Set_Method_Name(1, "PUSH");
  Set_Method_Name(2, "BEGIN");
  Set_Method_Name(3, "PULL");
}                                                // void Set_Method_Names(0)
```

```
private void Add_Actions() {
    States  tmp_state;
    Choices tmp_choice;

    // Primary
    // tmp_state = State(PRIMARY, 1);                    // State 1 (Initial)
    // IDENTIFY is not included - therefore this state is not needed

    tmp_state = State(PRIMARY, 2);                        // State 2 (Idle)
    tmp_state.Add_x_Choices(8);
    tmp_choice = tmp_state.Choice(1);                   // State 2, Choice 1
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(8), 0, REC, UNI, NONE));
    tmp_choice.Add_Action(new Msg(Message_Name(4), 1, SND, UNI, COMMIT));
    tmp_choice.Add_Action(new Msg(Message_Name(5), 1, REC, UNI, COMMIT));
    tmp_choice.Set_Next_State(3);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(2);                   // State 2, Choice 2
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(7), 0, REC, UNI, NONE));
    tmp_choice.Add_Action(new Msg(Message_Name(10), 1, SND, UNI, NONE));
    tmp_choice.Add_Action(new Msg(Message_Name(11), 1, REC, UNI, NONE));
    tmp_choice.Set_Next_State(5);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(3);                   // State 2, Choice 3
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(6), 0, REC, UNI, NONE));
    tmp_choice.Add_Action(new Msg(Message_Name(14), 1, SND, UNI, NONE));
    tmp_choice.Add_Action(new Msg(Message_Name(15), 1, REC, UNI, NONE));
    tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
    tmp_choice.Set_Next_State(8);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(4);                    // State 2, Choice 4
    tmp_choice.Add_Action(new Msg(Message_Name(17), 1, REC, UNI, NONE));
    tmp_choice.Set_Next_State(14);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(5);                    // State 2, Choice 5
    tmp_choice.Add_Action(new Msg(Message_Name(21), 1, REC, UNI, NONE));
    tmp_choice.Set_Next_State(6);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(6);                    // State 2, Choice 6
    // no further action, txn completed successfully
    tmp_choice.Add_Action(new Msg(Message_Name(22), 1, REC, UNI, NONE));
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
    tmp_choice.Set_Next_State(2);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(7);                    // State 2, Choice 7
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
    tmp_choice.Set_Next_State(2);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(8);                   // State 2, Choice 8
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
    tmp_choice.Set_Next_State(2);
    //******************************************************************//
```

```
tmp_state = State(PRIMARY, 3);                              // State 3 (Begun)
tmp_state.Add_x_Choices(2);
tmp_choice = tmp_state.Choice(1);                         // State 3, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, SND, UNI, ABORT));
tmp_choice.Set_Next_State(16);
//******************************************************************//
tmp_choice = tmp_state.Choice(2);                         // State 3, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
// needed for start-up Primary
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(4);
//******************************************************************//

tmp_state = State(PRIMARY, 4);                   // State 4 (sub-state of 3)
tmp_state.Add_x_Choices(2);
tmp_choice = tmp_state.Choice(1);                       // State 4, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(16);
//******************************************************************//
tmp_choice = tmp_state.Choice(2);                         // State 4, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, SND, UNI, ABORT));
tmp_choice.Set_Next_State(16);
//******************************************************************//

tmp_state = State(PRIMARY, 5);                            // State 5 (Enlisted)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                         // State 5, Choice 1
// need to specify action that has to be taken to decide whether
// PREPARE or COMMIT has to be sent; so far undefined action used ![tss,1.8.97]
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//******************************************************************//
tmp_choice = tmp_state.Choice(2);                         // State 5, Choice 2
// for the time being, the PRIMARY should only abort in PREPARED state
// therefore introduced a wrong control message
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
// expect message from Filter - not really needed
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
```

```
//*******************************************************************//
tmp_choice = tmp_state.Choice(3);                     // State 5, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(10);
//*******************************************************************//

tmp_state = State(PRIMARY, 6);                        // State 6 (Prepared)
tmp_state.Add_x_Choices(4);
tmp_choice = tmp_state.Choice(1);                     // State 6, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
// expect message from Filter - not really needed
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                     // State 6, Choice 2
// invalid !!
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(30), 3, REC, UNI, ABORT));
tmp_choice.Set_Next_State(7);
//*******************************************************************//
tmp_choice = tmp_state.Choice(3);                     // State 6, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(15);
//*******************************************************************//
tmp_choice = tmp_state.Choice(4);                     // State 6, Choice 4
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(30), 3, REC, UNI, COMMIT));
tmp_choice.Set_Next_State(7);
//*******************************************************************//

tmp_state = State(PRIMARY, 7);                          // State 7 (Error)
tmp_state.Add_x_Choices(1);
tmp_choice = tmp_state.Choice(1);                     // State 7, Choice 1
tmp_choice.Set_Next_State(0);
//*******************************************************************//

tmp_state = State(PRIMARY, 8);          // State 8 (Enlisted - Secondary)
tmp_state.Add_x_Choices(6);
tmp_choice = tmp_state.Choice(1);                     // State 8, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Change_Role("PRIMARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
```

```
tmp_choice = tmp_state.Choice(2);                    // State 8, Choice 2
// need to specify action that has to be taken to decide whether
// PREPARE or COMMIT has to be sent .. for the time being an undefined
// one will be used [tss,1.8.97]
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Change_Role("PRIMARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 8, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(12);
//*****************************************************************//

tmp_choice = tmp_state.Choice(4);                    // State 8, Choice 4
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Change_Role("PRIMARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(5);                    // State 8, Choice 5
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
// message not needed but easier to implement...
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Change_Role("PRIMARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(6);                    // State 8, Choice 6
//tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), READONLY ???));
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(29), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Change_Role("PRIMARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//

tmp_state = State(PRIMARY, 9);        // State 9 (Prepared - Secondary)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                    // State 9, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, REC, UNI, ABORT));
```

```
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                     // State 9, Choice 2
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 3, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                     // State 9, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(30), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Set_Next_State(7);
//*****************************************************************//

tmp_state = State(PRIMARY, 10);                   // State 10 (substate of
tmp_state.Add_x_Choices(5);                       // 5, Enlisted)
tmp_choice = tmp_state.Choice(1);                 // State 10, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(29), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 10, Choice 2
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                   // State 10, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(5), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(9), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(4);                   // State 10, Choice 4
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(28), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(4), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(11);
//*****************************************************************//
tmp_choice = tmp_state.Choice(5);                   // State 10, Choice 5
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(28), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(4), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(11);
//*****************************************************************//

tmp_state = State(PRIMARY, 11);                    // State 11 (substate of
tmp_state.Add_x_Choices(2);                        // 6, Prepared)
```

```
tmp_choice = tmp_state.Choice(1);                 // State 11, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(15);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                 // State 11, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, SND, UNI, ABORT));
tmp_choice.Set_Next_State(15);
//*******************************************************************//

tmp_state = State(PRIMARY, 12);                 // State 12 (substate of
tmp_state.Add_x_Choices(2);                     // 8, Enlisted-Secondary)
tmp_choice = tmp_state.Choice(1);                   // State 12, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(28), 2, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(9);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                 // State 12, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//

// State 13 - RECOVERY - not completely evaluated
tmp_state = State(PRIMARY, 13);                       // State 13
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                   // State 13, Choice 1
tmp_choice.Add_Action(new Log(Log_Name(5), R, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(17), 1, SND, UNI, NONE));
tmp_choice.Set_Next_State(14);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 13, Choice 2
tmp_choice.Add_Action(new Log(Log_Name(4), R, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(3);                 // State 13, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//

tmp_state = State(PRIMARY, 14);                     // State 14 (Idle)
tmp_state.Add_x_Choices(2);
tmp_choice = tmp_state.Choice(1);                   // State 14, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(19), 1, SND, UNI, NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, NONE));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
```

```
tmp_choice = tmp_state.Choice(2);                    // State 14, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(18), 1, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(20), 1, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//

tmp_state = State(PRIMARY, 15);                     // State 15 (Prepared)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                    // State 15, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(26), 3, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 15, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(5), 3, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(9), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 15, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(24), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//

tmp_state = State(PRIMARY, 16);                     // State 16 (Begun)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                    // State 16, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(26), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 16, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(5), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(9), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 16, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//


// Secondary
//tmp_state = State(SECONDARY, 1);                    // State 1 (Initial)
// Identification process is not included - therefore this state
//is not needed
```

```
tmp_state = State(SECONDARY, 2);                         // State 2 (Idle)
tmp_state.Add_x_Choices(7);
tmp_choice = tmp_state.Choice(1);                    // State 2, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(4), 1, REC, UNI, NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(8), 0, SND, UNI, NONE));
// dummy log-entry to avoid null-pointer crash [tss,16.8.97]
tmp_choice.Add_Action(new Log(Log_Name(1), W, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(5), 1, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(3);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                    // State 2, Choice 2
tmp_choice.Add_Action(new Msg(Message_Name(7), 1, REC, UNI, NONE));
// dummy log-entry to avoid null-pointer crash [tss,16.8.97]
tmp_choice.Add_Action(new Log(Log_Name(1), W, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(8), 1, SND, UNI, NONE));
tmp_choice.Set_Next_State(4);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 2, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(10), 1, REC, UNI,NONE));//PUSH
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(7), 0, SND, UNI, NONE));
// dummy log-entry to avoid null-pointer crash [tss,16.8.97]
tmp_choice.Add_Action(new Log(Log_Name(1), W, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(11), 1, SND, UNI, NONE));
tmp_choice.Set_Next_State(5);
//*****************************************************************//
tmp_choice = tmp_state.Choice(4);                    // State 2, Choice 4
tmp_choice.Add_Action(new Msg(Message_Name(14), 1, REC, UNI,NONE));//PULL
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(6), 0, SND, UNI, NONE));
// dummy log-entry to avoid null-pointer crash [tss,16.8.97]
tmp_choice.Add_Action(new Log(Log_Name(1), W, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(15), 1, SND, UNI, NONE));
tmp_choice.Add_Action(new Change_Role("PRIMARY", NONE));
tmp_choice.Set_Next_State(8);
//*****************************************************************//
tmp_choice = tmp_state.Choice(5);                    // State 2, Choice 5
tmp_choice.Add_Action(new Msg(Message_Name(20), 1, REC, UNI, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(21), 1, SND, UNI, NONE));
tmp_choice.Set_Next_State(6);
//*****************************************************************//
tmp_choice = tmp_state.Choice(6);                    // State 2, Choice 6
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(7);                    // State 2, Choice 7
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//

tmp_state = State(SECONDARY, 3);                        // State 3 (Begun)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                    // State 3, Choice 1
```

```
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24) , 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//******************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 3, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//******************************************************************//
tmp_choice = tmp_state.Choice(3);                   // State 3, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(4);
//******************************************************************//

tmp_state = State(SECONDARY, 4);               // State 4 (substate of 3)
tmp_state.Add_x_Choices(2);
tmp_choice = tmp_state.Choice(1);                   // State 4, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//******************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 4, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//******************************************************************//

tmp_state = State(SECONDARY, 5);                    // State 5 (Enlisted)
tmp_state.Add_x_Choices(6);
tmp_choice = tmp_state.Choice(1);                   // State 5, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//******************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 5, Choice 2
// need to specify action that has to be taken to decide whether
// PREPARE or COMMIT has to be sent .. for the time being an undefined
// one will be used [tss,1.8.97]
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
```

```
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 5, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(12);
//*******************************************************************//
tmp_choice = tmp_state.Choice(4);                    // State 5, Choice 4
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(5);                    // State 5, Choice 5
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
// receive message from Filter not necessary but easier to implement
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(6);                    // State 5, Choice 6
//tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), READONLY ???);
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(29), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//

tmp_state = State(SECONDARY, 6);                     // State 6 (Prepared)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                    // State 6, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
tmp_choice = tmp_state.Choice(2);                    // State 6, Choice 2
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 3, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*******************************************************************//
```

```
tmp_choice = tmp_state.Choice(3);                    // State 6, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(30), 3, REC, UNI, ABORT));
tmp_choice.Set_Next_State(7);
//*****************************************************************//

tmp_state = State(SECONDARY, 7);                       // State 7 (Error)
tmp_state.Add_x_Choices(1);
tmp_choice = tmp_state.Choice(1);                    // State 7, Choice 1
tmp_choice.Set_Next_State(0);
//*****************************************************************//

tmp_state = State(SECONDARY, 8);          // State 8 (Enlisted - Primary)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                    // State 8, Choice 1
// need to specify action that has to be taken to decide whether
// PREPARE or COMMIT has to be sent .. for the time being an undefined
// one will be used [tss,1.8.97]
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 2, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                    // State 8, Choice 2
// need to specify action that has to be taken to decide whether
// PREPARE or COMMIT has to be sent .. for the time being an undefined
// one will be used [tss,1.8.97]
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
// expect message from Filter - not really needed but easier to implement
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 8, Choice 3
tmp_choice.Add_Action(new Msg(Message_Name(27), 2, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(10);
//*****************************************************************//

tmp_state = State(SECONDARY, 9);         // State 9 (Prepared - Primary)
tmp_state.Add_x_Choices(4);
tmp_choice = tmp_state.Choice(1);                    // State 9, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
// message from Filter not needed but easier to implement
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
```

```
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                    // State 9, Choice 2
// invalid choice !!!
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(30), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Set_Next_State(7);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                    // State 9, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(11);
//*****************************************************************//
tmp_choice = tmp_state.Choice(4);                    // State 9, Choice 4
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(30), 3, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Set_Next_State(7);
//*****************************************************************//

tmp_state = State(SECONDARY, 10);                // State 10 (substate of
tmp_state.Add_x_Choices(4);                      // 5, Enlisted - Primary)
tmp_choice = tmp_state.Choice(1);                 // State 10, Choice 1
tmp_choice.Add_Action(new Msg(Message_Name(29), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 10, Choice 2
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, REC, UNI, ABORT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                   // State 10, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(28), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(11);
//*****************************************************************//
tmp_choice = tmp_state.Choice(4);                   // State 10, Choice 4
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(3), ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(28), 2, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(11);
//*****************************************************************//
```

```
tmp_state = State(SECONDARY, 11);                // State 11 (substate of
tmp_state.Add_x_Choices(2);                      // 9, Prepared-Primary)
tmp_choice = tmp_state.Choice(1);                 // State 11, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(25), 3, SND, UNI, COMMIT));
tmp_choice.Add_Action(new Msg(Message_Name(26), 3, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, COMMIT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                 // State 11, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(23), 3, SND, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 3, REC, UNI, ABORT));
tmp_choice.Add_Action(new Change_Role("SECONDARY", NONE));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//

tmp_state = State(SECONDARY, 12);                // State 12 (substate of
tmp_state.Add_x_Choices(2);                      // 5, Enlisted)
tmp_choice = tmp_state.Choice(1);                   // State 12, Choice 1
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, REC, UNI, COMMIT));
tmp_choice.Add_Action(new Log(Log_Name(5), W, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(28), 2, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(6);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                 // State 12, Choice 2
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, REC, UNI, ABORT));
tmp_choice.Add_Action(new Msg(Message_Name(24), 2, SND, UNI, ABORT));
tmp_choice.Add_Action(new Log(Log_Name(3), W, ABORT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(4), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//

tmp_state = State(SECONDARY, 13);                   // State 13 (Recover)
tmp_state.Add_x_Choices(3);
tmp_choice = tmp_state.Choice(1);                   // State 13, Choice 1
tmp_choice.Add_Action(new Log(Log_Name(5), R, NONE));
tmp_choice.Add_Action(new Msg(Message_Name(17), 1, SND, UNI, NONE));
tmp_choice.Set_Next_State(14);
//*****************************************************************//
tmp_choice = tmp_state.Choice(2);                   // State 13, Choice 2
tmp_choice.Add_Action(new Log(Log_Name(4), R, COMMIT));
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(1), 0, SND, UNI, COMMIT));
tmp_choice.Set_Next_State(2);
//*****************************************************************//
tmp_choice = tmp_state.Choice(3);                 // State 13, Choice 3
tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
tmp_choice.Set_Next_State(2);
```

```
    //******************************************************************//

    tmp_state = State(SECONDARY, 14);                      // State 14 (Idle)
    tmp_state.Add_x_Choices(2);
    tmp_choice = tmp_state.Choice(1);                   // State 14, Choice 1
    tmp_choice.Add_Action(new Msg(Message_Name(18), 1, REC, UNI, NONE));
    tmp_choice.Set_Next_State(2);
    //******************************************************************//
    tmp_choice = tmp_state.Choice(2);                   // State 14, Choice 2
    tmp_choice.Add_Action(new Msg(Message_Name(19), 1, REC, UNI, NONE));
    tmp_choice.Add_Action(new Ctrl(Ctrl_Name(2), 0, SND, UNI, ABORT));
    tmp_choice.Set_Next_State(2);
    //******************************************************************//

  }                                                    // void Add_Actions(0)

  protected void Display_Stats(int n) {
    // not used so far
  }                                                  // void Display_Stats(1)

}                                                          // class Table_TIP

/*-------------------------- End of Table_TIP.java -------------------------*/
```

B.2 Secondary.java

The Primary class is similar to the Secondary class. I have therefore refrained from listing the code of that class.

```
/*
** Author : Tanja Schulz <tss@dcs.ed.ac.uk>, 09/07/97
** Files  : Secondary.java, Secondary.class
** Update : 11/06/97
** Package: simdcp
*/

package eduni.simdcp;

import eduni.simjava.*;
import eduni.simanim.*;

/* Methods defined in final class Secondary
** PUBLIC
** Secondary(String, int, int, int, String)
**        void    body()
**
** PROTECTED
**        void    Init_Variables()
**        int     Site()
**
** PRIVATE
**        void    Create_Lists()
**        boolean Handle_Start_Txn()
**        boolean Handle_End_Txn()
**        void    Begin_Txn_SECONDARY(int)
**        int     Decide_Choice_SECONDARY(States)
**        int     Handle_Choice_SECONDARY(Choices)
**        int     Handle_Action_SECONDARY(Actable)
**        int     Handle_Action_Msg_SECONDARY(Msg)
**        int     Handle_Action_Ctrl_SECONDARY(Ctrl)
**        int     Handle_Action_Log_SECONDARY(Log)
**        int     Handle_Action_Role_SECONDARY(Change_Role)
**        int     Handle_Failure_SECONDARY()
**        Message Receive_Message()
**        void    Send_Message(Msg)
**        void    Schedule_Message(Message)
*/
final class Secondary extends Site implements Constants {

  private Sim_port   port_site;                          // Port to site filter
  private Sim_port   port_primary;

  int primary_id;                           // Secondary sends/receives messages
  int secondary_at_site;                           // directly to/from 1 Primary
  int fail_state;

  private Site_Lists Lists;
```

```
// list of Secondaries that should fail
private static int[] fail_sec;

public Secondary(String name, int id, int site, int primary_id,
                 int x, int y, String i_gif, String p_gif) {
    // Constructor for Secondary at site
    super(name, id, SECONDARY, x, y, i_gif, p_gif);

    Debug("Initiating Secondary at site "+this.Id());
    this.secondary_at_site = site;

    this.port_site    = new Sim_port("port_Site"+site, p_gif,
                                     Anim_port.BOTTOM,
                                     Option.Port_Pos(SECONDARY, site));
    this.port_primary = new Sim_port("port_Primary"+primary_id, p_gif,
                                     Anim_port.TOP,
                                     Option.Port_Pos(SECONDARY, primary_id));
    add_port(port_site);
    add_port(port_primary);

    add_param(new Anim_param("State", Anim_param.VALUE, "",
              Option.State_Pos_x(SECONDARY, id), Option.State_Pos_y(SECONDARY, id)));
    add_param(new Anim_param("Status", Anim_param.STATE, "sleep", 0, 0));
}                                                    // constructor Site(2)

protected void Init_Variables() {
  // Method to initialise the class variables because SimAnim does not
  // destroy and recreate the objects.
    super.Init_Variables();
    primary_id = UNKNOWN;                        // no Primary assigned
    fail_state = 0;

    // list of secondaries to fail - must be hardcoded at the moment !!!
    // set index 0 = no. of sites to fail, index == site_id; if YES then fail
    fail_sec = new int[Option.Num_Sites() + 1];
    fail_sec[0] = 0;
    int i;
    for (i=0; i <= Option.Num_Sites(); i++) {
      // sites to fail
      //if (i == 10) {
        fail_sec[i] = YES;
        fail_sec[0]++;
        //}
        //else {
        //fail_sec[i] = NO;
        //}
    }

    this.Create_Lists();
    super.Set_Status(SLEEPING);
}                                                  // void Init_Variables(0)

public void body() {
    // Body for Secondary at site
```

```
    int start_state_num = 2;          // changed start_state_num from 1 to 2
    boolean stop;

    this.Init_Variables();

    stop = this.Handle_Start_Txn();

    while (!stop) {
        if (Is_Secondary()) {
            super.Show_Window();
            Info("-> Started Secondary "+Id());
            super.Set_Status(RUN_OK);
            this.Begin_Txn_SECONDARY(start_state_num);
        }
        else {
            super.Set_Status(SLEEPING);
            super.Hide_Window();
        }
        stop = this.Handle_End_Txn();
    }
    if (super.Status() == FAILED) {
        Info("<- Secondary "+Id()+" recovered after TXN completed");
    }

    Info("<- Global decision: "+Global.Name_CorA(super.Global_Decision()));

    if (super.Local_Decision() != ABORT) {
        Info("<- Terminated Secondary "+Id()+": "
                                +Global.Name_CorA(super.Global_Decision()));
    }
    else {
        Info("<- Terminated Secondary "+Id()+": "
                                 +Global.Name_CorA(super.Local_Decision()));
    }
}                                                          // void body(0)

private void Create_Lists() {
    // Method to create the lists for a Secondary
    this.Lists = new Site_Lists(SECONDARY, this.Id(), this.Site_Window());
}                                                  // void Create_Lists(0)

private boolean Handle_Start_Txn() {
    // Method to handle the start of a txn - initiates Secondary if required
    // Returns true if any error occured and txn not started
    int     info;
    boolean stop = true;
    Message msg;

    msg = this.Receive_Message();
    info = msg.Info();

    if ((msg.Round() == -1) && (msg.Name().compareTo("PARTICIPATE") == 0)) {
        if ((info == COMMIT) || (info == ABORT)) {
            super.Set_Role_Of_Site(SECONDARY);
```

```
            super.Set_Local_Decision(info);
        }
        else {
            super.Hide_Window();
        }
        stop = false;
    }
    else {
        Error("First message does not start txn");
        stop = true;
    }
    super.Store_Current_Msg(null);
    return stop;
}                                              // boolean Handle_Start_Txn(0)

private boolean Handle_End_Txn() {
    // Method to handle the end of a txn - either terminate entity
    // Should never be called unless a Secondary is set up but excluded
    // from txn.
    // Returns true if site terminated
    boolean stop = false;
    Message msg, new_msg;

    while (!stop) {
        msg = this.Receive_Message();

        if ((msg.Round() == -1) && (msg.Name().compareTo("END-TXN") == 0)) {
            // Transaction terminated
            super.Set_Global_Decision(msg.Info());
            stop = true;
            if ((super.Global_Decision() != UNKNOWN)
                                || (super.Local_Decision() == ABORT)) {
                Info("<- TXN completed.");
            }
            Info("<- Local decision: "
                             +Global.Name_CorA(super.Local_Decision()));
        }
        else {
            super.Store_Current_Msg(null);
        }
    }
    return stop;
}                                                // boolean Handle_End_Txn(0)


private void Begin_Txn_SECONDARY(int state_num) {
    // Wait for/send messages as detailed in table, starting in state given.
    int     next_state_num;
    boolean stop;
    States  cur_state;

    stop = false;

    while ((state_num <= Table().Total_States(SECONDARY)) && (!stop)) {
```

```
        super.Set_State_Num(state_num);
        if (state_num == Table().Recover_State(SECONDARY)) {
            // hold causes unpredictable behaviour in SimAnim - smr
            super.Set_Status(RUN_OK);
            sim_hold(RECOVER_DELAY);
        }

        cur_state = Table().State(SECONDARY, state_num);
        next_state_num = this.Decide_Choice_SECONDARY(cur_state);

        Debug("in state "+state_num);
        Debug("next state "+next_state_num);
        if (next_state_num == 0) {
            stop = true;                      // No state to move to so stop
        }
        else if (next_state_num == FAIL) {
            Debug("failed choices in state "+state_num);
            Error("<- TERMINATED BEFORE COMPLETION");
            stop = true;
        }
        else {
            state_num = next_state_num;
        }
    }
}                                              // void Begin_Txn_Secondary(1)

private int Decide_Choice_SECONDARY(States state) {
    // Method to select a choice of actions given a state.
    // If there are no choices or only one then there is no decision to
    // be made, otherwise the choice will be made depending on the first
    // action of each choice.  It may be simpler to try the first choice,
    // then if its first action fails the second choice could be tried.
    // If a first action succeeds then it is committed to that choice.
    // The next state will be returned (from the choice selected) or zero
    // to stop the txn.
    int next_state_num;
    int total_choices;
    int choice_num;

    Debug("Deciding choice for part"+this.Id());
    total_choices = state.Total_Choices();

    if (total_choices == 0) {                        // No choices available
        Error("no choices, stop");
        next_state_num = FAIL;
    }
    else {                  // This will try each choice until it can do one
        next_state_num = RETRY;
        choice_num = 1;
        while (next_state_num == RETRY) {
            if (choice_num > total_choices) {          // If no choices left
                next_state_num = FAIL;
            }
            else {                                             // Try choice
```

```
                Debug("trying choice "+choice_num);
                super.Set_Choice_Num(choice_num);
                next_state_num =
                  this.Handle_Choice_SECONDARY(state.Choice(choice_num));
            }
            choice_num++;
        }
    }
    return next_state_num;               // return next state number or fail
}                                           // int Decide_Choice_Secondary(1)

private int Handle_Choice_SECONDARY(Choices choice) {
    // Method to handle the choice of actions and return the next state.
    int action_num;
    int success;
    int next_state_num;

    action_num = 1;
    success = OK;

    while ((action_num <= choice.Total_Actions()) && (success == OK)) {

        success = Handle_Action_SECONDARY(choice.Action(action_num));

        if (super.Status() == FAILED) {
            Debug("action "+action_num+" not completed due to failure");
            success = Handle_Failure_SECONDARY();
        }
        else {
            if ((success == OK)) {
                             // First action okay so committed to this choice
                Debug("action "+action_num+" completed successfully");
                action_num++;
            }
            else if (action_num == 1) {
                // Could not handle first action so return to try
                // another choice
                // Leave stored message for use in trying next choice
                success = RETRY;
            }
            else {             // Handled first action so committed to choice
                success = FAIL;
            }
        }
    }
    if ((success == OK) && (super.Status() != FAILED)) {
        next_state_num = choice.Next_State();
    }
    else {
        next_state_num = success;
    }
    return next_state_num;                              // Return next state
}                                           // int Handle_Choice_SECONDARY(1)
```

```
private int Handle_Action_SECONDARY(Actable action) {
    // Method to handle the Secondary's action depending on what it is,
    // i.e. message to be sent/received, control action, log r/w.
    int success;

    success = FAIL;

    Debug("handling action - "+action.Display());

    if (action instanceof Msg) {                        // Action is message
        success = Handle_Action_Msg_SECONDARY((Msg) action);
    }

    else if (action instanceof Ctrl) {
        success = Handle_Action_Ctrl_SECONDARY((Ctrl) action);
    }

    else if (action instanceof Log) {
        success = Handle_Action_Log_SECONDARY((Log) action);
    }
    // needed for entities to change display from S to P !! [tss,19.8.97]
    else if (action instanceof Change_Role) {
        success = this.Handle_Action_Role_SECONDARY((Change_Role) action);
    }

    else {
        Error("unknown class");
    }
    Debug("returning success "+success);
    return success;
}                                           // int Handle_Action_Secondary(1)

private int Handle_Action_Msg_SECONDARY(Msg msg_act) {
    // Method to handle a Secondary's message action.
    // i.e. protocol 'real' message to be sent or received.
    int       success;
    boolean   fail_point;
    Message   msg;

    success = FAIL;
    fail_point = false;

    if (msg_act.Send()) {                                     // Send message

        this.Send_Message(msg_act);

        success = OK;
    }

    else if (msg_act.Receive()) {                          // Receive message
        Debug("expecting message "+msg_act.Name()+","+msg_act.Round());

        msg = this.Receive_Message();
```

```
fail_point = super.Determine_Failure();

// only specific secondaries may fail
if (   (fail_sec[this.secondary_at_site] == YES) && (fail_point)
    && (fail_state != super.State_Num())) {
    Info("  Adversary, should I fail?");
    Info("      (before receiving "+msg.Name()
         +" in state "+super.State_Num()+":"
         +super.State_Name()+")");
    Info("  Adversary: YES");
    super.Set_Status(FAILED);

    // save state to check whether Secondary did already
    // fail in the same state before - don't fail because otherwise
    // it ends up in an endless loop !
    fail_state = super.State_Num();
    this.Send_Message(msg_act);
    success = OK;
    super.Store_Current_Msg(null);
    // count no. of message hops - basically the same as message
    // rounds but add one in case of a failure and count this also
    // for subsequent sites
    super.Inc_Local_Msg_Hops();
    // set global no. of message hops so that every primary can access it
    Global.Set_Sec_Message_Hops(secondary_at_site,super.Local_Msg_Hops());
}
else {
    Debug("  Adversary, should I fail?");
    Debug("      (before receiving "+msg.Name()
         +" in state "+super.State_Num()+":"
         +super.State_Name()+")");
    Debug("  Adversary: NO");


    if (msg.Src_Type() == PRIMARY) {
      if ((msg.Name().compareTo(msg_act.Name()) == 0)
                          && (msg_act.Round() == msg.Round())) {
        // Matching message pair - as expected
        success = OK;
        // Set Primary ID from first message
        primary_id = msg.Src_Id();
        Debug("Primary's Id: "+primary_id);

        super.Set_Global_Decision(msg.Info());
      }
      else {             // Not a pair - rec message was not expected
        Debug("expecting message "
                           +msg_act.Name()+","+msg_act.Round());
        Debug("received message "+msg.Name()+","+msg.Round());
      }
    }
    else if (msg.Src_Type() == SECONDARY) {
      Error("cannot handle Secondary->Secondary messages");
    }
```

```
            else if (msg.Src_Type() == SITE) {
              // Message from Site - should be control one
            }
            else {
              Error("Unknown Src_Type Handle_Action_Msg_SECONDARY(1) "+msg.Src_Type());
            }

            if (success == OK) {
              // Finished with message if it was expected
              // Otherwise, must store for remaining choices
              super.Store_Current_Msg(null);
            }
        }
    }
    else {
        Error("can't handle message");
    }

    return success;
}                                       // int Handle_Action_Msg_SECONDARY(1)

private int Handle_Action_Ctrl_SECONDARY(Ctrl ctrl_act) {
    // Method to handle a Secondary's Control action.
    // i.e. vote, timeout, commit or abort (end)
    int       success;
    Msg       msg_tmp;
    Message   msg;

    success = FAIL;

    if (ctrl_act.Name().compareTo("VOTE") == 0) {
        // Control action is to vote, done by checking info field
        if (ctrl_act.Info() == super.Local_Decision()) {
            success = OK;
            Info("Voting "+Global.Name_CorA(super.Local_Decision()));
        }
        else {
            success = FAIL;
            Debug("failing vote ctrl message");
        }
    }

    else if (ctrl_act.Name().compareTo("TIMEOUT") == 0) {
        // Control action is to expect a Timeout message
        msg = Receive_Message();

        if (msg.Name().compareTo("TIMEOUT") == 0) {
            if (ctrl_act.Msg().Round() == msg.Round()) {
                success = OK;
                Info("TIMEOUT in state "+super.State_Name());
            }
            else {
                Error("failing timeout, ctrl msg rnd "
```

```
                    +ctrl_act.Msg().Round()+", msg rnd "
                    +msg.Round()+" - not current");
                Error("local message round is "+super.Local_Msg_Round());
            }
        }
        else {
            Debug("Message received not TIMEOUT");
        }
        if (success == OK) {
            super.Store_Current_Msg(null);
        }
    }

    else if (   (ctrl_act.Name().compareTo("COMMIT") == 0)
             || (ctrl_act.Name().compareTo("ABORT")  == 0)) {

        if (ctrl_act.Is_Msg()) {
            // A commit or abort control action must involve
            // sending or receiving a message
            msg_tmp = ctrl_act.Msg();

            if (msg_tmp.Receive()) {
                msg = Receive_Message();

                if ((msg.Name().compareTo("END-TXN") == 0)
                                    && (msg.Round() == -1)
                                    && (msg.Info() == ctrl_act.Info())) {
                    success = OK;
                }
                else if (ctrl_act.Name().compareTo(msg.Name()) == 0) {
                    // Message does not end txn - may be for another choice
                    success = OK;
                    super.Store_Current_Msg(null);
                }
            }
            else {
              msg = new Message(ctrl_act.Name(),SITE,secondary_at_site,null,
                                SECONDARY,Id(),0,ctrl_act.Info(),null,true);
              this.Schedule_Message(msg);
              success = OK;
            }
        }
        else {
            Error("invalid Ctrl action for commit or abort, no message");
       }
    }
    else if (   (ctrl_act.Name().compareTo("PULL") == 0)
             || (ctrl_act.Name().compareTo("PUSH")  == 0)
             || (ctrl_act.Name().compareTo("BEGIN")  == 0)) {

      if (ctrl_act.Is_Msg()) {

        msg_tmp = ctrl_act.Msg();
```

```
        if (msg_tmp.Receive()) {
          msg = Receive_Message();
          Debug("Ctrl message receive - invalid action");
        }
        else {
          msg = new Message(ctrl_act.Name(),
                            SITE, secondary_at_site, null,
                            SECONDARY, this.Id(), 0, ctrl_act.Info(),
                            null, true);
          this.Schedule_Message(msg);

          success = OK;
        }
      }
      else {
        Error("unknown ctrl message");
      }
    }
    else if (ctrl_act.Name().compareTo("END") == 0) {

      msg = new Message("END-TXN",SITE,secondary_at_site,null,SECONDARY,Id(),
                        -1, super.Global_Decision(), null, true);
      this.Schedule_Message(msg);
      success = OK;
    }
    return success;
}                                     // int Handle_Action_Ctrl_SECONDARY(1)

private int Handle_Action_Log_SECONDARY(Log log_act) {
    // Method to handle a Secondary's log actions.
    // i.e. log read or write.
    int       success, info;
    String    name;
    Log       log_entry;

    success = FAIL;

    if (log_act.RW() == W) {
        // Write log information to DT log
        this.Lists.Write_DT_Log(log_act);
        Info("DT Log write: "+log_act.Name()
             +" ("+Global.Name_CorA(log_act.Info())+")");
        success = OK;
    }
    else if (log_act.RW() == R) {
        // Read last DT log entry and match with expected log action
        log_entry = this.Lists.Read_DT_Log();
        name = log_entry.Name();
        info = log_entry.Info();
        if (log_act.Name().compareTo(name) == 0) {
            Info("DT Log read: "+name+" ("+Global.Name_CorA(info)+")");
            Global.Inc_DT_Log_Reads();
            success = OK;
        }
```

```
        else {
            Debug("log entry ("+log_act.Name()+", "+log_act.Info()
                  +") expected, read ("+name+", "+info+")");
        }
    }
    else {
        Error("Unknown log access");
    }
    return success;
}                                       // int Handle_Action_Log_SECONDARY(1)

// needed for entities to change display from S to P !!! [tss,19.8.97]
private int Handle_Action_Role_SECONDARY(Change_Role role_act) {
  int success;

  success = FAIL;

  if (role_act.Name() == "PRIMARY") {
    super.Set_Status(CHANGE_PRIM);
    super.Set_Role_Of_Site(PRIMARY);
    success = OK;
  }
  else if (role_act.Name() == "SECONDARY") {
    super.Set_Status(RUN_OK);
    super.Set_Role_Of_Site(SECONDARY);
    success = OK;
  }
  else {
    Error("Invalid Role !");
  }
  return success;
}                                      // int Handle_Action_Role_PRIMARY(1)


private int Handle_Failure_SECONDARY() {
    // Method to handle failure by waiting for a message and trying to
    // recover.
    int     last_msg_count;
    int     next_state;
    boolean recover_point;
    Message msg, last_msg = null;

    recover_point = false;
    next_state = -1;
    last_msg_count = 0;

    if (super.Status() != FAILED) {
        Error("Status is not failed - should not be handling failure");
    }

    while (next_state == -1) {

        msg = Receive_Message();
        if (last_msg_count == 0) {
```

```
        Debug("Message is first, reset last message");
        last_msg = msg.Clone();
        last_msg_count++;
    }
    else {
        Debug("Name: "+last_msg.Name()+", "+msg.Name()+" - Type: "
              +last_msg.Src_Type()+", "+msg.Src_Type()+" - Id: "
              +last_msg.Src_Id()+", "+msg.Src_Id());
        if (   (last_msg.Name().compareTo(msg.Name()) == 0)
            && (last_msg.Src_Type() == msg.Src_Type())
            && (last_msg.Src_Id() == msg.Src_Id())         ) {
            // Message is same as last
            last_msg_count++;
            Debug("Message is same, count "+last_msg_count);
            last_msg = msg.Clone();
        }
        else {
            last_msg_count = 0;
            Debug("Message is different, reset last message");
            last_msg = null;
        }
    }

    if (last_msg_count >= DETECT_MESSAGE_LOOP) {
        // If message cycle detected then force recovery
        Info("");
        Info("*** Detected a dependency loop with "
             +Global.Short_Type(msg.Src_Type())+msg.Src_Id()+" \t***");
        Info("*** Possibly resulting from non-recovery \t***");
        Info("*** Message "+msg.Name()+" received "+last_msg_count
             +" \t***");
        Info("*** Protocol has probably blocked \t***\n");
        Info("*** Forcing recovery to break cycle \t***\n");
        Global.Inc_Timeout_Cycles();
        recover_point = true;
    }
    else {
        // Message does not indicate cycle so allow choice
        recover_point = super.Determine_Recovery();
    }
    if (recover_point) {
        Info("  Adversary, should I recover?");
        Info("      (before receiving "+msg.Name()+")");
        Info("  Adversary: YES");
        // DO NOT CHANGE STATUS - HANDLED LATER
        next_state = Option.Table().Recover_State(SECONDARY);
        if (msg.Name().compareTo("RECOVER") == 0) {
          super.Store_Current_Msg(null);
        }
    }
    else {
        Debug("  Adversary, should I recover?");
        Debug("      (before receiving "+msg.Name()+")");
        Debug("  Adversary: NO");
```

```
            if (!msg.Control()) {
                // Received a protocol message so send a timeout back.
                // Will only send a timeout to coordinator (not used!).
                // Assumes that Secondary only sent HELP-ME, no reply sent
                if ((msg.Src_Type() == COORD)
                            && (msg.Src_Id() == super.Coord_Id_Of_Part())) {
                    msg = new Message("TIMEOUT",
                                      COORD, super.Coord_Id_Of_Part(), null,
                                      SECONDARY, this.Id(), msg.Round(),
                                      msg.Info(), null, true);
                    this.Schedule_Message(msg);
                    // Current message is lost otherwise loop occurs
                    // May need the message after recovery - unsure
                    super.Store_Current_Msg(null);
                }
                else {
                    Debug("Protocol message "+msg.Name()+" from "
                          +Global.Short_Type(msg.Src_Type())+" ignored");
                    super.Store_Current_Msg(null);
                }
            }
            else {
                if (msg.Name().compareTo("END-TXN") == 0) {
                    next_state = 0;
                }
                else {
                    Debug("Control message "+msg.Name()+" from "
                      +Global.Short_Type(msg.Src_Type())+" ignored");
                    super.Store_Current_Msg(null);
                }
            }
        }
    }
    return next_state;
}                                           // int Handle_Failure_SECONDARY(0)

private Message Receive_Message() {
    // Method to wait for a message (event) or return the first queued
    // in buffer.  The buffered message will be returned first.
    Message   msg = null, reply_msg = null;
    Sim_event ev = new Sim_event();

    msg = super.Stored_Message();

    if (msg != null) {              // Only wait for message if not one stored
        Debug("using stored message");
    }
    else {
        Debug("No message currently stored");
        Debug("number of events queued: "+sim_waiting());
        Debug("waiting for event to arrive");

        sim_get_next(ev);
```

```
        msg = (Message) ev.get_data();

        Trace_Msg_Received(msg);

        super.Store_Current_Msg(msg);            // Store current message

        if (msg.Control()) {
            Global.Inc_Ctrl_Msgs_Rec();
            Debug("control msgs received "+Global.Ctrl_Msgs_Rec());
        }
        else {
            Info("Received "+msg.Name());
            Global.Inc_Real_Msgs_Rec();
            Debug("real msgs received "+Global.Real_Msgs_Rec());
        }
    }
    if (msg.Round() == (super.Local_Msg_Round() + 1)) {
        super.Inc_Local_Msg_Round();
    }
    return msg;
}                                            // Message Receive_Message(0)

private void Send_Message(Msg msg_act) {
    // Method to send a message given the source type and action
    // describing message.
    int[]   part_list = new int[MAX_SITES+1];
    int[]   dest_list = new int[MAX_SITES+1];
    Message msg;

    Debug("about to send message");

    if (msg_act.Round() == (Global.Message_Round() + 1)) {
        Global.Inc_Msg_Round();
    }
    if (msg_act.Round() == (super.Local_Msg_Round() + 1)) {
        super.Inc_Local_Msg_Round();
    }
    // added new Status for TIP [tss,25.7.97]
    if ((super.Status() == RUN_OK) || (super.Status() == CHANGE_PRIM)) {
        if (msg_act.Unicast()) {
            msg = new Message(msg_act.Name(),
                              PRIMARY, primary_id, null,
                              SECONDARY, this.Id(), msg_act.Round(),
                              msg_act.Info(), null);
            this.Schedule_Message(msg);
        }
        else {
            // not used !!
            Debug("only multicast msg is HELP-ME");
            if (super.Stored_Message() != null) {
                Debug("Assuming can recover from stored msg");
                // Otherwise have to ask Coord, assuming it hasn't failed
            }
            else {
```

```
                    Error("must multicast HELP-ME to Parts");
                    // Either to Parts only or also Coord
                }
            }
        }
        else if (super.Status() == FAILED) {
            msg = new Message("TIMEOUT",
                              PRIMARY, primary_id, null,
                              SECONDARY, this.Id(), msg_act.Round(),
                              msg_act.Info(), null, true);
            this.Schedule_Message(msg);
        }
        else {
            Error("Can't send messages in any other status");
        }
    }                                                   // void Send_Message(1)

    private void Schedule_Message(Message snd_msg) {
        // Method to actually send a message to intended recipient.
        Message msg;

        msg = snd_msg.Clone();

        if (msg.Dest_Type() == SITE) {

        sim_schedule(port_site, LINK_DELAY, SITE, msg);
        Trace_Msg_Sent(msg, port_site.get_pname());
        }
        else {
          sim_hold(2.0);
          sim_schedule(port_primary, LINK_DELAY, PRIMARY, msg);
          Trace_Msg_Sent(msg, port_primary.get_pname());
        }

        if (msg.Control()) {
            Global.Inc_Ctrl_Msgs_Sent();
            Debug("control msgs sent "+Global.Ctrl_Msgs_Sent());
        }
        else {
            Info("Sent "+msg.Name()+" (state "+super.State_Num()
                 +":"+super.State_Name()+")");
            Global.Inc_Real_Msgs_Sent();
            Debug("real msgs sent "+Global.Real_Msgs_Sent());
            // Assume if message sent then no fail/recover cycle exists
            super.Reset_Times_Failed();                        // New state
        }
    }                                               // void Schedule_Message(1)

    protected final int Site() {
      // Method to return the Site Id of the Secondary
      return this.secondary_at_site;
    }                                                             // int Site(0)
}                                                            // class Secondary
/*-------------------------- End of Secondary.java --------------------------*/
```