TOKEN-BASED PEER-TO-PEER INTERACTION COORDINATION

By

Theodore Chiasson

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA
JULY, 2003

Your file Votre référence
ISBN: 0-612-83700-9
Our file Notre référence
ISBN: 0-612-83700-9

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "Token-based Peer-to-peer Interaction Coordination" by Theodore Chiasson in partial fulfillment for the degree of Doctor of Philosophy.

Dated: April 8, 2003

External Examiner: [redacted]

Research Supervisor:

Examining Committee:

Departmental Representative:

DALHOUSIE UNIVERSITY

Date: **July, 2003**

Author: **Theodore Chiasson**

Title: **Token-based peer-to-peer interaction coordination**

Department: **Computer Science**

Degree: **Ph.D.** Convocation: **October** Year: **2003**

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

For Sandra, Zachary, Kali, and Jacob.

Contents

List of Figures

Abstract

The rapid emergence of Electronic Commerce has caused a so-called "digital divide" to form between those with access to the Internet and those without. While governments have tried to lessen this divide through the introduction of public access points and infrastructure subsidies, little is being done to address the barriers to entry for functionally illiterate and cognitively impaired populations. Our research attempts to address this issue by individualizing end-user interactions with computerized systems on a domain-specific basis. We take the view that each end-user's system should act as a peer in the computing environment, allowing end-users to own and control their information. Shifting from the current client-server computing environment to the peer-to-peer paradigm of computing should facilitate the customization of interactions that these populations require while maintaining their privacy. Existing applications programming environments are geared towards the client-server model of computing. Research into peer-to-peer application development has indicated that the process-oriented programming environment facilitates some aspects of peer-to-peer application development, but introduces challenges with respect to distributed data management. Existing blocking distributed transaction coordination mechanisms rely on global state to facilitate centralized control of distributed resources, but there is no concept of global state in the process-oriented model. This thesis introduces a new model of token-based peer-to-peer interaction coordination (the TPIC model). This new model maintains consistency in a peer-to-peer environment without relying on global state.

Chapter 1

Introduction

This thesis introduces a new model of transaction coordination in a peer-to-peer environment, called the token-based peer-to-peer interaction coordination (TPIC) model. In this new model, it is assumed that peers establish contracts before participating in a peer-to-peer interaction. A mobile, disposable token is used to coordinate peer-to-peer interactions, and actions to be performed at each of the peers are serialized to form a token plan. Each time a token visits a peer, a microtransaction in the token's plan is executed. The execution of a microtransaction involves movement of data from containers on the token into the peer's database, performance of a database transaction which moves the peer's database state from one consistent state to another consistent state, and finally, movement of data from the peer's database to containers on the token. The peer-to-peer interaction holds locks in the peer's database only while the token is present, yielding a non-blocking model. Although the microtransactions are executed serially among peers for a single interaction, high concurrency among independent interactions should be facilitated due to the non-blocking nature of the coordination model.

1.1 Motivation

This thesis forms part of a larger body of work, the Knowledge Acquiring Layered Infrastructure (KALI) project at Dalhousie University, which aims to increase accessibility for functionally illiterate and cognitively impaired populations [9]. Our research to date has indicated that accessibility for these populations requires extensive individualization of interactions, which in turn requires additional mechanisms for ensuring privacy. In this section, we introduce the motivation and context for the KALI project and provide a brief description of the paradigm shift we anticipate will be necessary to increase accessibility for these populations.

1.1.1 Privacy

Increasing accessibility of systems through personalization of end-user interactions requires extensive personal information about the end-user to be stored in the system with appropriate protection and access control mechanisms. Individuals will not allow this information to be collected and stored unless they can retain ownership and control over it. To allow end-users to own and control their information, we propose a paradigm shift from the client-server architecture of computing to the peer-to-peer architecture. Under the peer-to-peer paradigm, end-user personalization information such as medical records and financial information can be stored on systems under the end-user's control. This movement of personal information to end-users systems will allow processing of personalization information to occur locally under the end-user's control. Existing approaches to personalization require that each service maintain a copy of end-user information for each user of the service. This distribution of an end-user's personalization information across many sites reduces end-user control of end-user information.

A peer-to-peer application is a distributed application in which each peer can act as a client and/or as a server at the application level, allowing flexible assignment of processing responsibilities. In contrast, a client-server distributed application forces

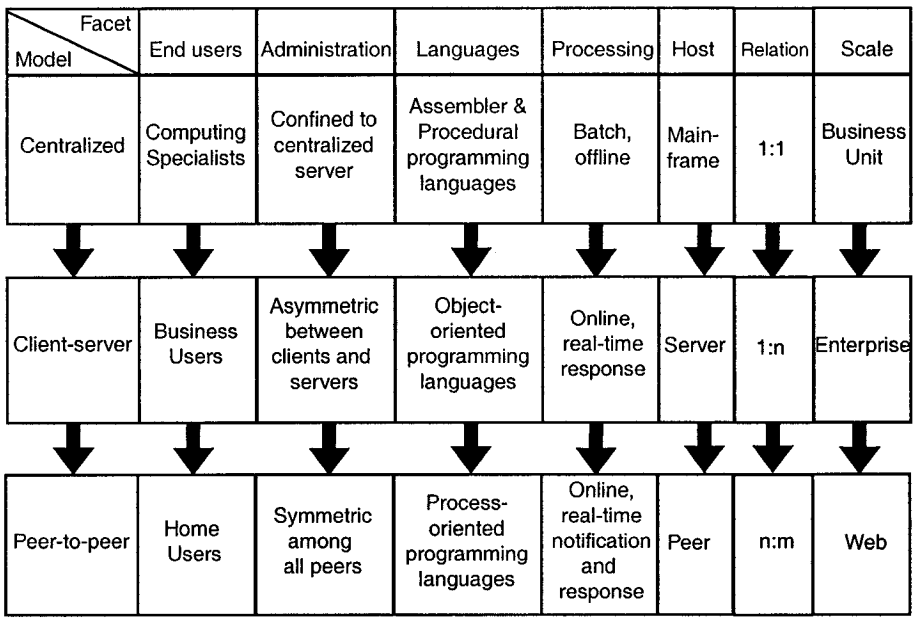

Figure 1.1: Shifting Paradigms.

the bulk of the processing onto the server. Processing involves access to data, and in the client-server architecture the data is therefore centralized at the server. In the peer-to-peer paradigm, data can be distributed across peers, and responsibility for processing of data can be assigned to the peer with access to the data. Interactions among peers typically involve some sharing of information, since the information to be processed is distributed among the peers.

The peer-to-peer paradigm involves peer-to-peer interactions at all levels, including the system, communication, and application levels [3]. We perceive that changing to a peer-to-peer paradigm will involve changes in many facets of computing, as depicted in Figure 1.1.

Moving to the peer-to-peer paradigm may be facilitated by a different application development environment than is currently employed, as peers in the system must operate independently *at the application layer.* Process-oriented application development environments (as described by Strom [40]) such as HERMES [41] appear well-suited to the development of peer-to-peer applications [3]. In a process-oriented

development environment, there is no concept of global state or time, leading naturally to process independence. This lack of global state, however, leads to data management challenges with respect to existing blocking distributed transaction processing models which require global state to ensure consistency among participants. This thesis provides a new model of peer-to-peer interaction coordination that does not rely on global state and that can be used to share data in a peer-to-peer environment without compromising privacy. Section 2.8 provides a detailed discussion of the process-oriented development environment.

1.1.2 From client-server to peer-to-peer

Over the past four decades, hardware systems have repeatedly been replaced by newer, faster, and less expensive components. This pattern of advancement in hardware systems, which was fueled in part by the proliferation of home computers and, more recently, hand-held wireless devices, has allowed hardware designers to start with a clean slate at each iterative pass, thus reducing the complexity of the design process. Systems and application software, on the other hand, have followed an evolutionary path of advancement where each new release maintains backwards compatibility with the earlier systems. Many software systems (e.g., IBM DB2, Oracle) that have evolved over a long period of time still have some original components from as far back as thirty and forty years ago. In addition, the functionality offered by software systems has increased steadily. Taken together, these factors have contributed to the ever-increasing complexity of software systems.

Development of software systems has traditionally been accomplished by software professionals for software professionals. Although the development group has evolved from a single "guru" into teams of programmers, the domain knowledge of these teams is still technology-focused. Lack of knowledge in application-specific domains has led to a computing environment that non-technical end-users find unwieldy to operate.

The dramatic increases in the number of computer users and the availability of high-speed network connectivity coupled with significantly reduced costs is causing a

societal shift towards a knowledge-based economy [15]. In fact, the number of end-users has risen so dramatically that a *revolutionary* approach to software development is becoming more feasible. There are sufficient numbers of new users to justify a market for domain-specific, specialized devices that do not follow an evolutionary pattern of development.

In order to increase accessibility, a shift from computing-centric to human-centric development of systems is required. In a human-centric environment, end-user interactions with the system are customized based on extensive personalization information. Domain experts, who are not computer scientists or computer engineers, participate in the definition of personal traits that can be used to customize interactions to carry out domain-specific tasks.

1.2 Overview of the TPIC model

Traditional blocking distributed database transaction coordination implementations require global state to ensure that distributed transactions move the database from one consistent state to another consistent state [7]. We anticipate the need for a peer-to-peer interaction coordination model that does not use global state. We propose a layered model for the coordination of peer-to-peer interactions, called the TPIC model. The layers of the TPIC model include the application layer, the contract layer, the application support layer, the interaction processing layer, the token processing layer, and the local DBMS processing layer. This section provides a high-level description of the TPIC model and of each of the layers in the model.

The TPIC model provides a mechanism for the controlled sharing of data across peers. Within the TPIC model, peer-to-peer applications generate domain-specific contracts between pairs of peers using the services of the contract layer. Each contract includes domain-specific terms describing the business rules and timing constraints that apply. Additionally, a contract contains meta-data detailing the remote schema information about data items to be accessed at each peer. Each peer extends their

database schema when they enter into a contract. This extension provides a storage area within the database used as a staging area for data that is transferred between peers. While the establishment of contracts is an important area of research, it is not the focus of this thesis, and the existing contract.net [38] approach to distributed negotiation, described in Section 2.10, will be adopted for this thesis.

The peer-to-peer application support layer uses established contracts to initiate an interaction that will accomplish a domain-specific task. The information in the contracts is used to generate an interaction plan. This interaction plan details the business tasks to be performed, and the order in which to perform them, in order to complete an interaction in accordance with the terms in the contracts governing the interaction. A token plan is generated by choosing one or more paths through the interaction plan. This token plan is passed as a parameter to the interaction processing layer of the TPIC model. The interaction processing layer builds a token based on the parameters it receives from the application support layer. A token includes a catalog, a token plan, a set of containers, and a log. The token catalog is used to facilitate access to data on each of the peers involved in the interaction, as well as access to data on the token itself. Information in the contracts held by the initiating peer is used to build the token catalog. The token plan indicates where the token should travel based on the status returned from each peer's DBMS (commit or abort), and what action should be taken at each peer. The peer that initiates the interaction builds the token plan based on the terms and constraints in its contracts. The schema information from each contract is used to construct the token's containers. Containers are used to store and move data among peers.

The steps in the token plan, termed microtransactions, correspond to subtransactions in traditional distributed transaction processing. The containers on the token can be seen as volatile storage areas, or buffers. In contrast to traditional models of distributed transaction coordination, each microtransaction in the token plan is executed in isolation and processed sequentially instead of in parallel at the different sites. Each microtransaction moves a single peer's database from one consistent

state to another consistent state. When a microtransaction starts, it first moves data from a container on the token into a temporary table in the peer's database. Upon completion, a microtransaction moves data from the temporary table in the peer's database onto a container on the token. The data in the temporary table is considered "stale" after the token departs. This "stale" information can, however, be used during recovery processing if the token is lost. Thus, when a token leaves a peer, the peer's database is left in a consistent state.

Recovery in the TPIC model must use compensating microtransactions, since each microtransaction execution performs a complete transaction on a peer's DBMS. When a microtransaction does fail for some reason, the token can be sent back to the originating peer for recovery processing. The application support layer on the originating peer can produce a new token plan that includes compensating microtransactions or alternative microtransactions to the one that failed, and processing can continue. If the token is lost, the initiating peer must request information from every peer that has executed microtransactions under the current interaction to try to rebuild the token's containers on the basis of the "stale" information found in the temporary tables at each participant. If insufficient information is available, the interaction layer will invoke arbitration and pass along the information it has gathered to the contract layer to have the problem resolved outside the scope of the model.

In this chapter, the motivation for the TPIC model was presented, together with an overview of the model. Chapter 2 provides background in areas related to this thesis, including transaction processing, advanced transaction coordination mechanisms, distributed transaction processing models, middleware, workflow systems, contracts, and peer-to-peer computing. In Chapter 3, the TPIC model is informally illustrated using several examples. A detailed description of the TPIC model is presented in Chapter 4, followed by proofs showing the correctness of this new model of transaction coordination in Chapter 5. Chapter 6 describes a preliminary implementation that validates the TPIC model. Chapter 7 presents the conclusions of the thesis and describes future research directions.

Chapter 2

Background and Related Work

Areas related to this research include distributed transaction coordination, concurrency control techniques, peer-to-peer computing, the process-oriented programming environment, transaction processing, business transactions, and automated contract negotiation. To a lesser extent, the areas of middleware and workflow systems are important to consider to ensure that the results of the research are relevant in today's computing environment. Background information on state of the art in each of these fields is described below in the context of this research.

2.1 Historical perspective on DBMS Processing

In 1970, Codd proposed the relational database model [11]. This simple model provided a theoretical framework for relational databases that was absent in the earlier models (e.g., hierarchical model, network model). The relational model provided a set-theoretic mathematical model for dealing with data stored in databases as tuples. Operations supported by the model included insertion, deletion, and update (which can be modelled as a deletion plus an insertion) of tuples. The Structured Query Language (SQL) was developed using set theoretic concepts [8], where the fundamental operations of the language are set operations (join, project, union, etc.).

Database management systems (DBMSs) were then constructed with two main

components - a front end or preprocessor for parsing SQL statements and generating the necessary read, write, and update commands to make the requested changes in database state, and a back end or data management component responsible for maintaining database state [14]. The front end accepts SQL statements in a FIFO order, and the execution of each SQL statement is expected to move the database state from one consistent state to another consistent state.

The DBMS was a relatively straightforward system to implement, since no concurrency control was required. A log of the actions performed by the SQL statement was generated, and in the event of a failure the log was used to UNDO the effects of the failed SQL statement. Thus, SQL statements executed in FIFO order with logging used to reverse the effects of failed executions exhibited four important properties with respect to the database: atomicity, consistency, isolation, and durability (ACID). Atomicity is the all or nothing property - either all of the updates occur or none of them occur. To ensure atomicity, logging is used to reverse the updates that occur before a failure. Consistency is the guarantee that the updates will move the database from one consistent state to another consistent state. Isolation requires that no intermediate results are visible - this is guaranteed by executing one SQL statement at a time. Durability is the property that once a statement has finished executing, the effects of the statement will be durable even in the event of a system crash. This property is guaranteed by flushing all changed database values to the disk before writing the log record that ends a statement's execution.

The sequential execution of individual SQL statements was found to be an inefficient use of system resources. To increase utilization, it was desirable to execute many requests in parallel. However, the arbitrary interleaving of reads and writes associated with distinct update operations can violate the isolation property and lead to a corrupt database state. While each individual SQL statement would move the database from one consistent state to another consistent state, the interleaving of actions from different SQL statements would provide no such guarantee. This has led to the development of transactions and serializability theory. Note that in this

context a single SQL statement can be though of as a complete transaction.

2.2 Transaction processing

A transaction is a sequence of one or more read and/or write actions (together with either a commit or an abort action) performed against a database which, taken together, bring the database forward from one consistent state to another consistent state. A set of transactions executed in serial order clearly maintains database consistency. However, the arbitrary interleaving of reads and writes from different transactions can lead to corruption of database state [5]. If, however, the reads and writes resulting from a given interleaving are equivalent to the reads and writes generated by some serial execution of the set of transactions, then database consistency will be preserved. This is the premise for research on serializability of transactions.

2.2.1 ACID transaction properties

While serializable schedules are guaranteed to maintain database consistency, the theory of serializability is not used in practice for validating transaction schedules. Instead, formal theoretical proofs have been validated which show that transaction histories which maintain certain properties (which are easier to accommodate than serializability verification) represent serializable schedules. These properties are atomicity, consistency, isolation, and durability. Taken together, these properties are referred to as the ACID transaction properties. If all transactions adhere to the ACID properties, the serializability of concurrently running transactions is guaranteed [5].

To ensure that the concurrent execution of transactions is equivalent to some serialized ordering of the individual transactions, the transaction processing coordinator ensures that the ACID transactional properties are enforced. The three basic approaches to enforcing the ACID transactional properties are locking, timestamping, and optimistic methods [4].

2.2.2 2-Phase Locking (2PL)

In the locking approach, a read or write lock is obtained on a data item before it is read or written. Multiple transactions can simultaneously hold read locks on the same data item, but write locks are held exclusively. In order to obtain a conflicting lock on a data item, a transaction must block until the conflicting lock on the item is released. In order to guarantee isolation, strict 2 phase locking (2PL) is used, where locks are obtained throughout the lifetime of a transaction but no locks are released until the transaction is aborted or committed [19]. One of the problems with locking is the potential for deadlock to occur, where two or more transactions have each locked a data item while waiting for the release of data locked by the other transaction.

2.2.3 Timestamping

In timestamping approaches, every data item is assigned a timestamp. The timestamp on an item is compared to that of the requesting transaction before the data item is accessed. If the transaction timestamp is older than the timestamp on the data item, the requesting transaction is aborted and restarted. Otherwise, the transaction is allowed to access the data item, and the data item's timestamp is updated. Thus no locking occurs, and there is no possibility of deadlock. Timestamping produces serializable execution histories, but the overhead associated with aborting transactions can be prohibitive [6].

2.2.4 Optimistic methods

For optimistic methods, it is assumed that conflicts are very rare. Transactions have three phases: read, validate, and write. During the read phase, writes are local to the transaction. The transaction is assigned a timestamp, and the validation phase ensures that no conflict has occurred. In the event of a conflict, the transaction is restarted. Otherwise, the write phase is used to commit the transaction. Performance of optimistic methods breaks down when conflicts are common [26].

Of these three basic approaches, strict 2PL is by far the most widely implemented concurrency control mechanism in production or prototype database processing systems. In 2PL, only (read, read) locks are compatible - that is, (read, write), (write, read), and (write, write) locks all cause conflicts. These conflicting locks sometimes cause transactions to block even though the transactions could execute concurrently without violating the semantic integrity of the database system.

2.3 Advanced transaction models

Advanced transaction models have been proposed which either replace or enhance 2PL in order to take advantage of the additional concurrency gains made possible when semantic constraints are used to allow transactions to continue in cases where they would block under 2PL. Some advanced transaction models are discussed below.

2.3.1 Nested transactions

Nested transactions, introduced by J. Moss in the early '80s [31], allow for the recursive definition of a tree of subtransactions. Child transactions can only start after their parent starts, and a parent transaction cannot complete until all of its children have completed. If a parent transaction aborts, all of its children must be aborted, but if a child transaction aborts the parent has a choice in the way it will proceed. The parent transaction can choose to abort, or it can invoke an alternative or contingency subtransaction in lieu of the failed subtransaction. Nested transactions permit increased modularity and a higher degree of intra-transaction concurrency as compared to non-nested transactions, while still maintaining the isolation properties of the global transaction.

2.3.2 Open nested transactions

In *open nested transactions*, also developed by Moss [30], the isolation requirement is relaxed by allowing the committed results of subtransactions to be viewed by other

nested transactions before the global transaction has committed. Only subtransactions that commute with the committed subtransaction are allowed to view its results. While only read activities commute in traditional systems, update operations can sometimes be defined as commutative. One example is the increment operation on a counter variable. Open nested transactions achieve a higher degree of inter-transactional concurrency.

2.3.3 Sagas

Sagas, introduced in 1987 by Hector Garcia-Molina [20], uses the concept of compensating transactions to allow a series of transactions to be executed as a group. A Saga consists of a series of ACID transactions $T_1, ..., T_n$ and a series of compensating transactions $C_1, ..., C_n$ where each C_i effectively compensates for the execution of the corresponding T_i. During the execution of the Saga, if all transactions commit the Saga is committed. If a transaction T_k aborts, then compensating transactions $C_{k-1}, ..., C_1$ are performed, causing a forward correction of the database. Sagas are useful for dealing with long-lived transactions.

2.3.4 Transaction chopping

By chopping transactions into smaller pieces and executing the pieces as transactions, the amount of time that locks are held can be reduced. The ACID properties of the transaction may be compromised by this action - in particular, the isolation property may be violated since committed pieces of transactions may be viewed by other transactions even though the transaction as a whole has not yet completed, and, indeed, may still abort. One approach to this problem is the concept of *transaction chopping* [35]. In this approach, a chopping graph is generated based on the syntactic constraints implicit in the transaction's structure. Any additional known semantic knowledge, such as commutativity of actions on a data item (i.e., a counter variable), may be used to reduce the number of constraints on the transaction. If parts of a transaction fail, they can be retried until they commit. The correctness criteria in

the chopping graph ensure that any interleaved execution of the chopped transaction with other concurrently running transactions (which might also have been chopped) will result in a serializable execution history. In the event of a system failure, the transaction log must maintain additional knowledge for the recovery mechanism, since parts of a chopped transaction may have committed while the global transaction has not yet completed.

2.4 Distributed Transaction Concurrency Control

A distributed database is a collection of logically integrated data distributed across systems that are connected by a computer network. A distributed database management system consists of software that makes the distributed aspects of the underlying data as transparent as possible. Transactions in a distributed database environment are differentiated on the basis of the location of the data they access. Local transactions occur on a single node and access the data in only one database, while any transaction that accesses data from more than one database is termed a global transaction.

The problem of ensuring that transactions executing concurrently in a distributed database produce serializable histories is similar to that of the single database case, with some additional complexity. In a distributed database, a global transaction is divided into subtransactions, and each subtransaction is submitted for processing at a single database. If two or more global transactions execute concurrently, the individual subtransactions that occur at the same database will be serialized by the local database transaction coordinator. However, different databases may generate a different serialization order for subtransactions, with the result that the overall global transactions are not serializable. Thus, in a distributed database system, both local and global serializability must be enforced. This involves both a distributed concurrency control technique and a distributed atomic commit protocol.

The approaches for maintaining serializability in a distributed database environment are based on extensions to the single database approaches of locking, timestamping, and optimistic methods.

2.4.1 Distributed 2PL

In distributed 2PL, all of the locks for the global transaction must be acquired before any of the local subtransactions can commit [4]. A coordinator uses the two-phase commit protocol (described in subsection 2.5.1) to ensure that all locks have been acquired (prepare phase) before any of the subtransactions commits. While the commit protocol introduces messaging overhead proportional to the number of participants in the distributed transaction, the number of participants in most distributed transactions is low since the way in which data is partitioned in a distributed database environment tends to place data on sites where it is typically accessed [4].

2.4.2 Distributed timestamping

In a distributed environment, there is no single clock which can be used to assign unique timestamps. Furthermore, events can occur simultaneously at two or more sites. Rather than attempt to synchronize clocks across sites, the concept of global time is introduced. This global time is defined as the concatenation of the local clock on the participant and a participant identifier. This local clock is not the local system clock, but rather a local counter that is advanced by the transaction processing protocol. Global time is maintained by advancing the local clock each time an event occurs on the local site (start of a transaction or receipt of a message). When a message is sent between two participants, the message timestamp is the timestamp of the sender. Upon receipt of a message, a participant will advance their local clock if the received message timestamp is greater than the local timestamp. This causes participants to synchronize with each other when they communicate. While clocks may drift apart when two participants do not communicate, they don't need to be synchronized since they are not collaborating or communications would be occurring.

Given that global time is available to each of the participants, timestamping can be enforced as in the centralized case [4].

2.4.3 Distributed optimistic approaches

As in the centralized case, it is assumed that conflicts are very rare. The validate phase is performed in two stages, a local validation stage and a global validation stage. If any local validations fail, the entire transaction is aborted. If all of the local validations succeed, a global validation phase ensures that all outstanding transactions with a smaller ID terminate before proceeding [4].

2.5 Distributed Transaction Coordination

2.5.1 Two-phase commit (2PC)

The two phase commit protocol is by far the most widely implemented approach to distributed transaction coordination in commercial and prototype distributed database systems (e.g., DB2, Oracle, Sybase). In the two-phase commit protocol, subtransactions of a distributed transaction either all commit or all abort. A centralized coordinator requests that all participants prepare to commit. In the "prepare" phase, each participant is asked to execute their part of the distributed transaction and "prepare" to commit. The participants then attempt to acquire all of the locks they will need to eventually commit. If any participant responds with an "abort", the distributed transaction is aborted and the coordinator issues an "abort" message to each participant. If, however, all participants respond with "prepared", the coordinator enters the "commit" stage. In this stage the coordinator logs that the transaction has been committed and issues a "commit" directive to each participant. Each participant then commits their part of the distributed transaction, and issues an acknowledgment to the coordinator that they have successfully committed. Once all of the participants have acknowledged a commit to the coordinator, the coordinator

logs the commit messages to stable storage and issues an "end transaction" directive and the transaction has been completed.

2.5.2 Three-phase commit (3PC)

The blocking nature of the 2PC algorithm can cause a transaction to block at all sites if a single site fails or a network partition occurs during certain phases of the 2PC protocol. The 3PC protocol addresses issues of node failure and network partitioning by moving to a three-phase protocol [37]. In the first phase, the coordinator requests that all participants vote for a commit or an abort outcome. All of the participants issue votes of commit or abort to the coordinator, ending the first phase. If all of the participants vote to commit, the coordinator issues the instruction to pre-commit. The pre-commit instruction informs all participants that they should prepare to commit. All of the participants then acknowledge that they received the prepare to commit instruction, ending the second phase of the protocol. Next, the coordinator issues the global commit directive, and each participant commits their part of the transaction and acknowledges their success to the coordinator. This ends the third and final phase of the protocol.

In the first phase, if any participant decides to abort they will vote abort and commence local abort processing. Upon receipt of one or more abort votes, the coordinator will instruct all participants to pre-abort. The pre-abort instruction informs participants that they should prepare to abort local processing of the transaction. All of the participants issue acknowledgments to the coordinator that they have received the pre-abort instruction, ending the second phase of the protocol. Finally, the coordinator issues a global-abort instruction to each of the participants. Each of the participants then initiates local abort processing and acknowledges to the coordinator that they have aborted the transaction. When all of the acknowledgments have been received, the transaction has been fully aborted.

2.6 Multidatabases

In a multidatabase, the databases are heterogeneous and each local DBMS is a pre-existing DBMS which cannot be changed in support of distributed processing. The need for multidatabase systems stems from the independent deployment of heterogeneous systems within an enterprise which must later be integrated, mergers and acquisitions involving companies that have deployed databases from different vendors, and interoperability issues for trading partners pursuing value chain integration.

In some multidatabase implementations, a layer of processing is added on top of each local DBMS, and data values are logically separated into local variables that only local transactions can access and global variables that global transactions can access [2]. Unlike the distributed database situation, each local DBMS executes transactions autonomously, and the global transaction manager cannot interfere or coordinate how local transactions (or subtransactions) are handled by the local DBMSs. Transactions that access both local and global data can conflict with each other transitively even if they access different data items at different sites. This forces the global transaction coordinator to assume that all global transactions conflict, since the global scheduler has no control on how the local DBMSs will schedule requests.

Unlike distributed transaction processing, sites may commit local subtransactions without confirmation that the global transaction is successful. If a global transaction fails after one or more of its local subtransactions has been committed, compensating local subtransactions must be used to restore globally consistent state.

2.7 Peer-to-peer

In the centralized model of computing, large platform-dependent applications were written by expert programmers and access to the system was limited to business applications. These systems were replaced by client-server systems over networked environments, with complex servers accessed by relatively simple client applications.

As computing power has migrated from centralized systems out closer to the individuals accessing systems, the type of end-users and the modes of access have changed considerably. The shift from centralized to client-server computing increased the number and type of end-users able to access computerized systems. It is anticipated that the shift from client-server to peer-to-peer computing will similarly increase accessibility of systems by allowing the end-user system to act as a peer in the environment and to customize interactions for the end-user in a personalized fashion.

As computing devices continue to proliferate and as networking technologies continue to deliver faster connections at reduced cost, a paradigm shift from the client-server model to the more general peer-to-peer model is likely to occur. Several commercial ventures have recently emerged that use peer-to-peer networking to transfer data among distributed peers using centralized lookup services and centralized control (e.g., KaZaa [25], GnuTella [21]). These systems have scalability problems due to the centralized lookup and control services.

Recent research on the Chord project at MIT [39] has produced a potentially viable distributed lookup service as an alternative to the centralized control regime. The Chord approach allows for completely decentralized service lookup based on cooperation among the participating peers. Conceptually, the peers are arranged in a circle with each peer assigned to a node number based on a hash of the peer's ID. Each peer maintains a list of pointers to its neighbors at distances of 1, 2, 4, ..., 2^n around the circle. Thus, given an index value to look up, a *chord* is traversed to quickly approach the node that stores the value associated with the index.

Another recent development in the decentralized peer-to-peer environment is the Simple Object Access Protocol (SOAP), an XML-based lightweight protocol for exchange of information in a decentralized, distributed environment. The SOAP protocol consists of an envelope that describes the message contents and how to process it, encoding rules for application-defined data types, and a convention for remote procedure call and response representation [43].

With the advent of client-server computing, sophisticated desktop computers replaced the dumb terminals of the centralized model. This shift from dumb terminals to desktop computers caused a dramatic increase in systems administration complexity. In the centralized model, the systems administration was confined to the centralized servers. In the client-server model, client machines require the performance of administrative tasks such as upgrades and patches to the operating system and client software, backups, network connectivity administration, and troubleshooting. Also, the inherent asymmetry of clients vs. servers increases overall systems administration complexity, since servers have different requirements than clients. The growth in systems administration complexity associated with client machines was severely underestimated, and resulted in a significant loss of productivity for end-users [24].

Moving to a peer-to-peer model of computing will shift more functionality and responsibility to the end-user's system. In order to prevent a further escalation of systems administration complexity, a different approach to software development and deployment may be required. The existing distributed computing infrastructure is based primarily on the client-server oriented middleware, with distributed applications typically written in object-oriented programming languages.

It is anticipated that the process-oriented programming environment will be better suited to the development of peer-to-peer applications than the object-oriented programming environment, as it allows data and functionality encapsulated with in a process to migrate freely among peer systems. While moving to a peer-to-peer paradigm will increase the amount of systems administration required on the end-user's system, the process-oriented programming environment coupled with the symmetry of peer systems should allow for more automation of administrative processes.

2.8 Process-oriented programming environment

In process-oriented computing, software is constructed from processes with typed interfaces. Each process encapsulates data and functionality, and processes can only

be accessed through their interfaces. Thus, the input ports of a process entirely determine its interface. Processes communicate through message channels that connect the output port of one process to a type-compatible input port of another process. Processes exchange messages via their typed input and output ports. A process does not know about the implementation details of other processes.

There is no concept of global state or global time in a process-oriented system, as all data and functionality are local to their owning process [40]. Without shared state, processes are independent of each other. This reduces the complexity associated with process migration and distributed application development [29]. The independence of processes reflects the peer relationship among processes, where each process can take on the role of client or server as need be.

In contrast, the object-oriented programming environment allows for shared state through constructs such as templates, class variables, inheritance, and public and private methods and data. Applications are developed using shared global state, and distribution is effected through use of middleware facilities that allow for location transparency for the programmer at the expense of run-time migration transparency for the end-user (see section 2.11).

The lack of global state and process independence inherent in the process-oriented programming environment as described by Strom [40] make it an attractive model for the development of distributed applications. In the CORDS project [3], difficulties were encountered for the development of data-intensive distributed applications using the process-oriented programming environment, as described below. In the absence of global state and global time, no known transaction coordination mechanism could be applied directly to the management of data distributed among processes. This forced each process requiring access to a database to encapsulate the entire database, and to move the entire database if other processes needed access to it.

The main goal of this thesis is to address the issue of transaction coordination in the absence of shared global state. We see this aspect of the process-oriented programming environment as a significant hurdle that must be cleared to facilitate the

development of peer-to-peer applications that can be used to personalize interactions in a way that preserves privacy for end-users. For a detailed description of the process-oriented programming environment, see for example [40, 41, 44].

2.9 Business transaction protocol

Business-to-business E-Commerce is increasing as businesses begin to automate their interactions with suppliers and trading partners. Existing transaction processing mechanisms are not well-suited to the requirements of automating business transactions among autonomous parties.

OASIS (The Organization for the Advancement of Structured Information Standards, [33]), a not-for-profit global consortium whose focus is to develop standards for E-business, has recently produced the Business Transaction Protocol (BTP) specification. BTP includes a set of specific messages that get exchanged between computer systems supporting an application, together with rules defining the meaning and use of these messages. The BTP specification is designed to facilitate business transactions among parties with pre-existing contractual agreements, where a business transaction is defined as:

"... a consistent change in the state of a business relationship between two or more parties. A business relationship is any distributed state held by the parties which is subject to contractual constraints agreed by those parties." [32].

The BTP approach assumes pre-existing contracts among the parties involved in a business transaction. Furthermore, each party using BTP has an Application Element and a corresponding BTP element. The messages exchanged in the BTP protocol are sent and received in parallel to the existing application message exchanges. The BTP serves to coordinate the effects of application actions among the parties. This is done using a two-phase exchange of BTP messages.

In BTP, one of the systems involved in a business transaction is assigned the role of coordinator. A software agent at the coordinator plays the role of Superior, and

a software agent at each of the other parties plays the role of Inferior. The BTP is centered on the relationship between a coordinator and one of the other parties involved in a business transaction. The basic approach is for the application at the coordinator to issue an application message to a service application at one or more parties involved in the overall business transaction. For each of the parties playing an Inferior role, the coordinator (acting in the Superior role) issues a BTP message. This message informs the Inferior node of the application message that was initiated. The Inferior nodes reply to the coordinator with a BTP message indicating their preparedness to accomplish provisional effects of the requested application service. Moving to a prepared state implies that the provisional effect has been accomplished and the Inferior node is ready to either commit the changes (resulting in the final effect) or reverse the changes (counter effect) based on the decision of the coordinator. Once the coordinator has received sufficient information concerning the preparedness of the Inferior participants, a consistent set of final effects is chosen and the coordinator issues BTP messages to all Inferior nodes indicating if they should confirm (final effect) or cancel (counter effect).

In addition to a single coordinator with Inferior participants, BTP also supports the use of each Inferior participant as a Superior to other Inferior participants, yielding a business transaction tree. Each node in the tree between the root and the leaves acts in a dual role, as an Inferior to its parent node and as a Superior (coordinator) to its child nodes.

BTP provides for two types of interaction - atoms and cohesions. An atom follows the all-or-nothing approach, where all participants must be confirmed for the business transaction to proceed (i.e., every Inferior participant has a veto power of the business transaction). A cohesion allows for the coordinator to decide on a set of participants to confirm (the confirm set). This confirm set can evolve during the life of the business transaction, and can even include new Inferior participants that are added due to the failure of some other Inferior participant to prepare.

Recovery in BTP uses presumed abort. Persistent information is required at both

Superior and Inferior participants at certain junctures in the protocol in order to ensure recovery in the presence of node failures.

2.10 The contract net protocol

The contract net protocol, originally proposed by Smith in 1980 [38], is a protocol for establishing contracts among distributed autonomous sites. This protocol does not assume a centralized coordinator, and allows each participant in the protocol to act in both a client role and a server role interchangeably. The steps involved in securing a contract between two parties include:

1. **Task Announcement:** This is best described as a request-for-proposals (or RFP). The site requiring a service broadcasts an abstract representation of the task's requirements to the network. Information in the broadcast includes a description of the task, the expected capabilities of the contractors, information that bids should contain, and a deadline for delivery of bids.

2. **Evaluation:** Sites that receive the Task Announcement and are not already occupied evaluate the announcement and decide whether or not to place a bid.

3. **Bidding:** Sites that decide to bid on the RFP submit their bids to the site that announced the task, including details of their ability to successfully perform the task.

4. **Awarding:** The site requiring service evaluates the bids it has received and awards the task to the most suitable node(s), and issues a cancel message to the sites that were rejected.

A site that bids on an RFP is free to act as a manager and issue its own RFP's from other nodes, which allows for subcontracting of services. Sites that both bid on RFPs and issue RFPs are effectively acting as both a client and a server interchangeably.

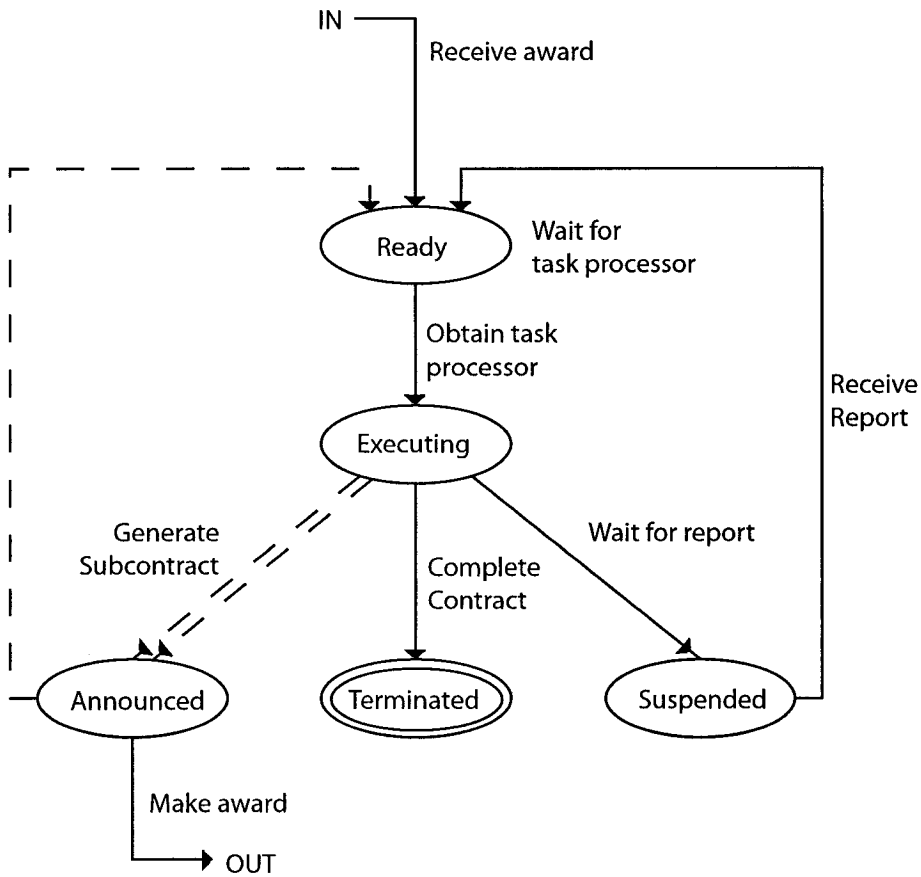

Figure 2.1: Structure maintained by a site for each task it has been assigned under the contract net protocol.

A site maintains a structure similar to that shown in Figure 2.1 for each of the tasks assigned to it.

In this thesis, the contract net protocol is used to perform rudimentary contract negotiation among peers. Contracts are established between peers before they enter into TPIC model interactions. While contract negotiation is an important requirement of the TPIC model, the methods involved in the negotiation of contracts are independent of the model and are not a focus of this thesis. The original contract net protocol provides sufficient services to support the basic contract negotiation requirements of the TPIC model.

2.11 Middleware

Middleware implementations such as CORBA [23], DCOM [12], and EJB [27] were developed in large part to solve problems of heterogeneity in distributed systems. Middleware can resolve issues of heterogeneity and facilitate coordination and communication of distributed components in the construction of distributed applications. The role of middleware in distributed application development is to present an abstract layer to the application programmer that hides the details of the underlying system infrastructure. Components in distributed systems need to communicate with other components, potentially on different hosts. This communication must occur over a network. Rather than program distributed systems by interfacing directly with the transport layer, middleware can be used to handle the session and presentation layers of the ISO network protocol stack.

Broad classes of middleware can be identified based on the primitives provided for the interaction between distributed components. The categories are transactional, message-oriented, procedural, and object or component based, and their primitives are, respectively, distributed transactions, message passing, remote procedure calls, and remote object requests [18]. Transactional middleware was developed to solve heterogeneity problems in distributed database applications. This type of middleware

lacks support for marshaling and has no well-defined interface for service offerings. Furthermore, the overhead associated with guaranteeing ACID properties is imposed even for applications that do not require transactional semantics. Message oriented middleware is good for distributed event notification architectures, but supports only at-least-once semantics. Furthermore, this type of middleware does not provide any support for marshaling or for synchronous communications. Procedural middleware addresses marshaling well, but lacks flexibility. All communications are one-to-one synchronous, only at-most-once semantics are supported, and the lack of location transparency restricts scalability. Object or component based middleware evolved from procedural middleware, and combines many of the best features of each of the other types of middleware. Object and component based middleware have proved their suitability for client-server oriented applications, but exhibit shortcomings with respect to new types of applications such as real-time, multimedia, and mobile applications [17]. Middleware presents a black-box style interface to the applications layer, and does not allow for quality of service provisioning or adaptability. Proposed solutions include frameworks which extend existing middleware functionality [36, 42] and adaptive or reflexive middleware implementations [13, 16].

The Web Services protocol stack replaces traditional middleware implementations with SOAP and XML over HTTP, yielding an extremely flexible model that lacks transactional support. The lack of support for eBusiness transactions in the Web Services model has resulted in a renewed interest in relaxed transactional semantics research [28, 34].

2.12 Workflow

A workflow is a computerized system that automates or facilitates a business process or part of a business process. Workflow management (WFM) systems have evolved from a number of different product areas including image-processing systems, document management systems, groupware applications, and project support software.

Workflow management systems have evolved to support the coordination of business processes. In this context, business processes are fairly coarse-grained activities, such as the invocation of an application or an action performed by an individual. In contrast to transaction processing systems, workflow systems do not typically guarantee ACID transaction properties [10].

The approach taken in the Workflow on Intelligent Distributed database Environment (WIDE) project [22] is to extend database functionality to support process-centric application environments such as workflow management systems. Database functionality is extended through additional advanced transaction support and high-level active rule support. Furthermore, a data support level (called the basic access layer) is introduced to shield implementations from the specific underlying database system in use.

The aim of the WIDE project was to develop an implementation of a next-generation workflow system with advanced transactional support. Since most businesses rely heavily on existing relational databases that do not provide advanced transactional support, a basic access layer was introduced to shield the workflow system from the underlying database management system. The basic access layer uses CORBA (see 2.11) to provide distributed access to the DBMS and to encapsulate relational data in an object-oriented way. This layer includes mapping support between the underlying relational model of the DBMS and the object-oriented model of the workflow environment.

Relational database transaction semantics are in general too restrictive to support the flexibility required by workflow systems. Workflows are process-oriented, and parts of a workflow may involve the execution of applications or the performance of manual operations by humans. Performance of traditional database systems relies on transactions with short duration to ensure minimal lock contention. Advanced transaction models have been proposed in the literature, but support for these models has not been added to mainstream databases. In the WIDE project, an advanced transaction support module is used to facilitate the needs of the workflow system.

Workflow models are extremely flexible, and it has been shown that workflow models form a superset of advanced transaction processing models [1]. In contrast to the WIDE project, the work by Alonso ([1]) shows how existing workflow systems (Flowmark in particular) can be used to create workflows that behave according to any desired advanced transaction processing model, thus eliminating the need for advanced transaction support at the database level. This approach, however, leads to additional complexity for the workflow designer, as generating workflows that support advanced transaction model semantics places the onus on the workflow designer to understand and implement advanced transaction semantics for each workflow they design.

Chapter 3

Informal description of TPIC model

In this chapter, the TPIC model is presented informally and illustrated through several examples. Chapter 4 provides a more detailed description of each layer in the TPIC model. In Chapter 5, proofs of correctness for TPIC interaction executions are presented. Chapter 6 describes algorithms for an early prototype implementation used to validate the TPIC model.

3.1 Introduction

Existing blocking distributed database transaction coordination implementations require global state to ensure that distributed transactions move the distributed database from one consistent state to another consistent state [7]. We anticipate the need for a peer-to-peer interaction coordination mechanism in support of personal ownership of end-user information that does not use global state and that allows peers to independently complete their part of a peer-to-peer interaction. To accomplish this we introduce a new model for the coordination of peer-to-peer interactions, termed the token-based peer-to-peer interaction coordination (TPIC) model, that does not rely on global state.

The TPIC model is made up of function layers. Layers in the model include the peer-to-peer application layer, the contract layer, the application support layer, the interaction layer, the token layer, and the local processing layer. Figure 3.1 shows the functional roles assigned to each layer and the data structures manipulated by each layer.

In the TPIC model, the actions to be performed in a peer-to-peer interaction are serialized on a token that is sent to each peer in turn. The TPIC model assumes that each peer involved in an interaction maintains a DBMS. The set of actions to be performed during a single visit to a peer is termed a microtransaction. If the execution of a microtransaction is successful, it will move a peer's database forward from one consistent state to another consistent state. If the execution of a microtransaction fails, it will have no effect on the peer's database. Microtransactions initiated by an interaction under the model will be viewed by each peer's DBMS in the same way as locally generated transactions. No software modifications to the peer's DBMS are necessary in support of the model.

A token has containers which are used to move data from one peer to another. The data in these containers is moved into the local database when the token arrives at a peer and moved back into the containers before the token leaves the peer. Thus, the data from the token containers is available during local processing, but is no longer available once the token has moved on. The results of the local processing are recorded in a log on the token. A token plan details the order in which peers will be visited by the token, and the actions to be taken at each peer. For each peer, the actions performed by a single microtransaction form a complete local transaction. The status code ("commit" or "abort") returned from a peer's DBMS is used to determine the next peer the token should visit based on the token plan. Each peer is assumed to have a uniquely identifying HostID (e.g., IP address).

Definition 3.1 A *microtransaction* is an ordered tuple (n, m, p), where n is the HostID of the peer which can execute the microtransaction, m is a reference to an SQL program at peer n, and p is a list of tables in token containers that should be

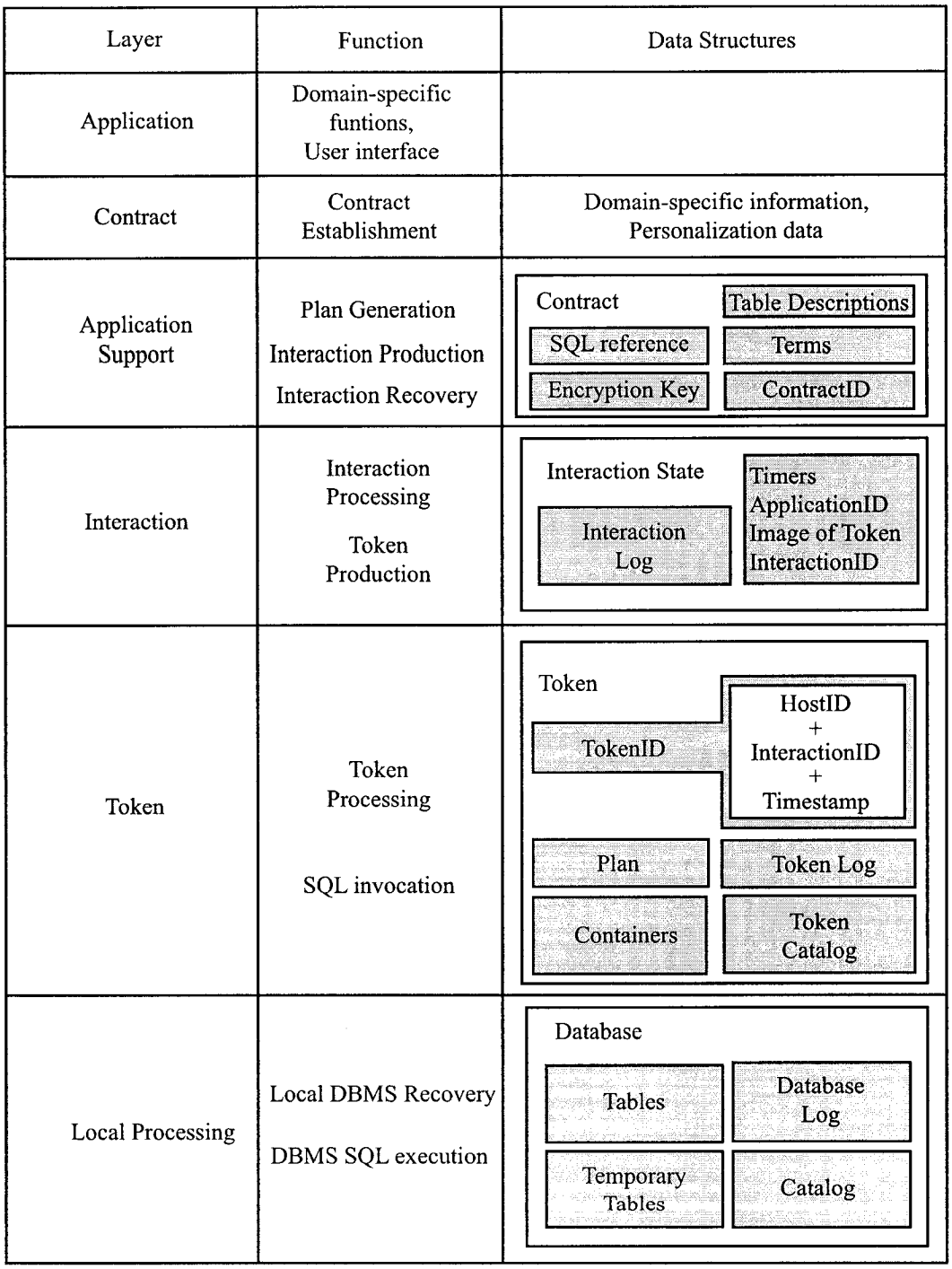

Figure 3.1: This figure shows the functions assigned to each layer of the token-based peer-to-peer interaction coordination model, and the data structures manipulated by each layer.

moved into n's database before executing the SQL program referenced by m, and moved back into token containers after the execution completes.

The peer that initiates a token-based interaction, termed the *originating peer* , is responsible for generating tokens and for invoking recovery if the token processing fails. Thus, the information needed to construct a token must be available to the originating peer before a peer-to-peer interaction can begin. Within the TPIC model, we assume that contracts have been established between pairs of peers. As a minimum, the originating peer must have an established contract with each of the peers involved in the interaction. Each contract includes references to SQL programs that reside on the peer where the program will be executed, as well as references to corresponding compensating SQL programs that reverse the effects of an SQL program's execution.

Contract terms in the TPIC model are a formal representation of the business rules governing interactions. These terms indicate the interdependencies between microtransactions to be executed during an interaction. For example, a business rule might state that delivery should not occur if payment is not received. Receipt of payment and activation of delivery are accomplished by the execution of two distinct microtransactions. Thus, the formal representation of this business rule in the contract would be the contract term (delivery, payment). Any interaction which adheres to this contract term would have to ensure that the delivery microtransaction is not performed if the payment microtransaction has not been successfully performed by the end of the interaction. From the customers viewpoint, it is important that the interaction ensure that delivery occurs if payment is received, resulting in the contract term (payment, delivery). The combination of these two contract terms effectively establishes an atomicity constraint between the two microtransactions payment and delivery.

When a contract is established, temporary tables are added to each peer's DBMS. These temporary tables are used to move data on and off of the token containers. The contract terms and additional information supplied by the peer-to-peer domain-specific application initiating the interaction and by the end-user profile maintained

by this application are used to establish an interaction plan.

3.2 Bank transfer example

As an example, consider the transfer of a sum of money from an account at bank A to an account at bank B. Using the TPIC model, a contract must have been established between A and B before an interaction can occur. Assume the business rules governing the transfer dictate an atomicity constraint among the three tasks in the interaction - withdraw from A, deposit to B, return a receipt to A. These three business tasks correspond to microtransactions W, D, and R, respectively. Thus, the contract terms would be $\{(W, D), (W, R), (D, W), (D, R), (R, W), (R, D)\}$.

Once a contract is in place, interactions can make use of the contract until it expires. An interaction will typically make use of multiple contracts, but for this simple example we will use only one. The application support layer uses the information in the contract and domain-specific constraints from the application to construct an interaction plan (see Figure 3.2). An interaction plan has the property that all paths from the source node to a sink node adhere to the terms of the contracts governing the interaction. A token plan is then selected from among the possible paths through the interaction plan (see Figure 3.3). This token plan is passed as a parameter to the interaction support layer, where a token is generated. Once started, an interaction will not terminate until a token plan has successfully completed. In this example, the token plan indicates that the token should execute a microtransaction at A (withdraw funds), then travel to B and execute a microtransaction (deposit funds, issue receipt), then return to A and execute a final microtransaction (store receipt) and terminate. If a code is returned from a peer's DBMS that is not handled by the token plan, the token will be returned to the originating peer for recovery processing.

When the token arrives at the token processing layer at A, all of the containers on the token contain empty tables. This causes the temporary tables at A associated with the contract to be initialized as empty. The SQL program referenced by the

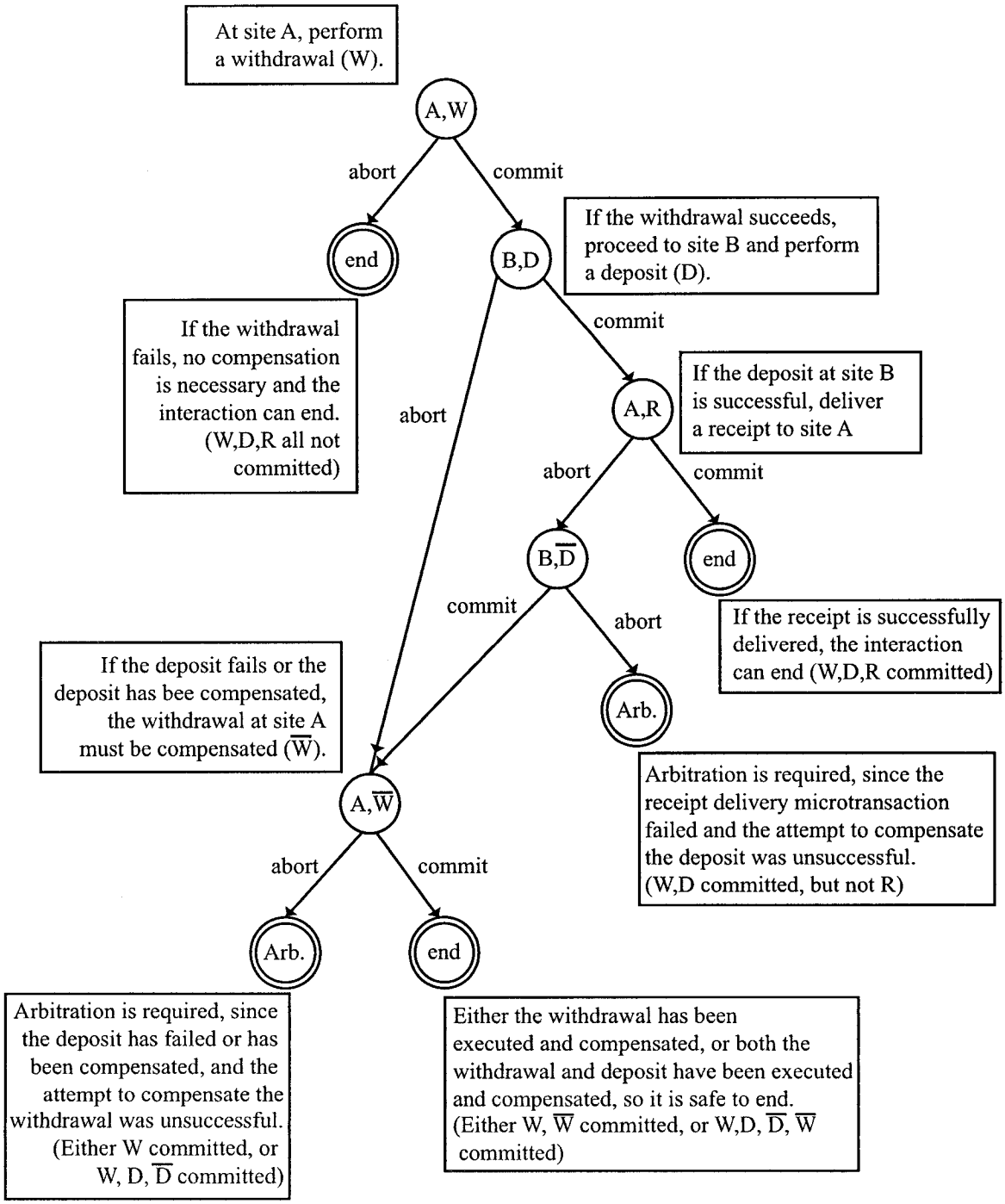

Figure 3.2: An interaction plan for the bank transfer example.

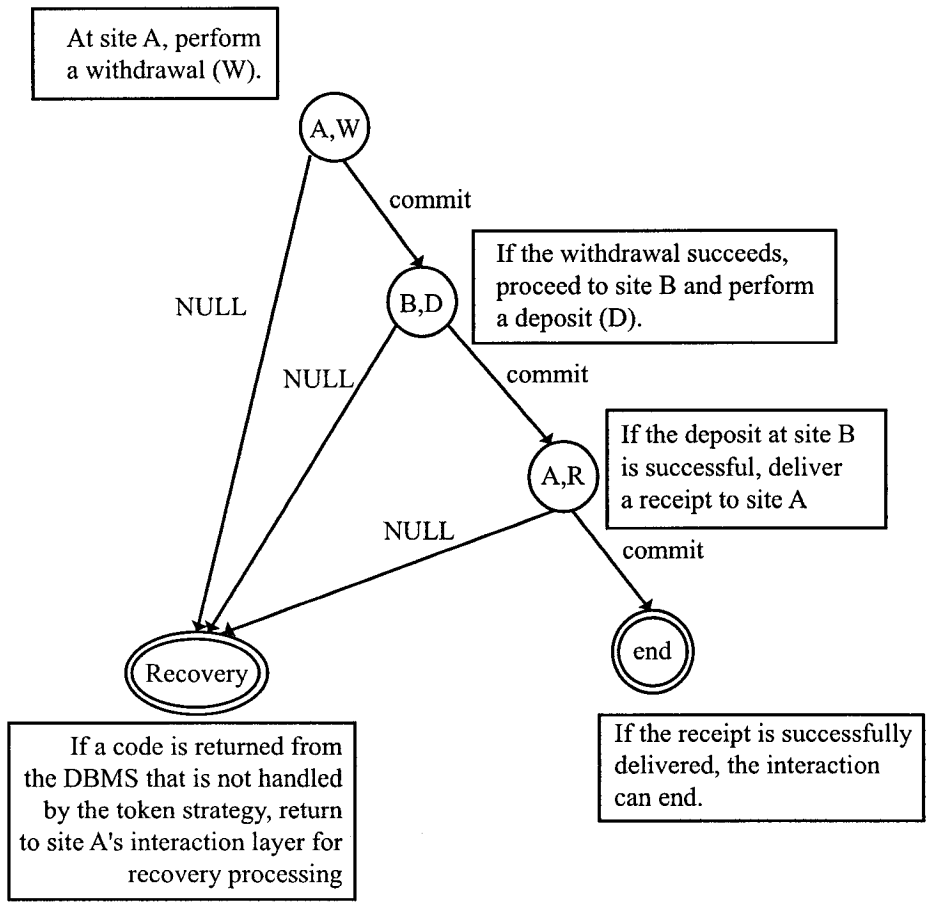

Figure 3.3: A simple token plan for the bank transfer example.

microtransaction is executed, moving funds from an account at A into a temporary table. The temporary tables are then moved back into the token containers. The content of the temporary tables at A is then considered "stale" - this content cannot be used for processing but may be needed for recovery purposes in the event of a lost token (see Section 4.3.5). The withdrawal from bank A to a container on the token effectively moves a sum of money out of a bank account at A, but the money has not yet been re-assigned to another account. In this context, the container on the token can be thought of as volatile storage that is used to move data among peers. If the withdrawal is successful, the token moves to bank B, where a deposit is made from the container on the token into the bank and a receipt is moved from B onto the token container (using the temporary tables in B's database). Finally, the token moves back to A, where the receipt is moved from the container on the token into A's database. Figure 3.4 illustrates the flow for this example.

In the TPIC model, the coordination role is placed on a mobile disposable token and the microtransactions in the peer-to-peer interaction are serialized. This allows each peer to complete a portion of the interaction independently. The actions performed at each peer consist of a sequence of SQL statements submitted as a transaction to the peer's DBMS. If the token plan fails to complete, the effects of the microtransactions that have completed may have to be compensated. Revisiting the bank transfer example above, if the deposit at bank B fails, the token will be sent back to the originating peer (A) and details of the token's actions as recorded in the token log will be handed to the application support layer. Based on the microtransactions that have been performed, a new token plan is produced that will either reverse the actions performed by the token through compensation or complete the interaction through re-tries or alternative actions.

Compensating microtransactions can also be embedded in the original token plan if failures are anticipated in advance. For example, the token plan can be extended with a compensating microtransaction to reverse the withdrawal from A if the microtransaction at B fails (see Figure 3.5). The original withdrawal can be successfully

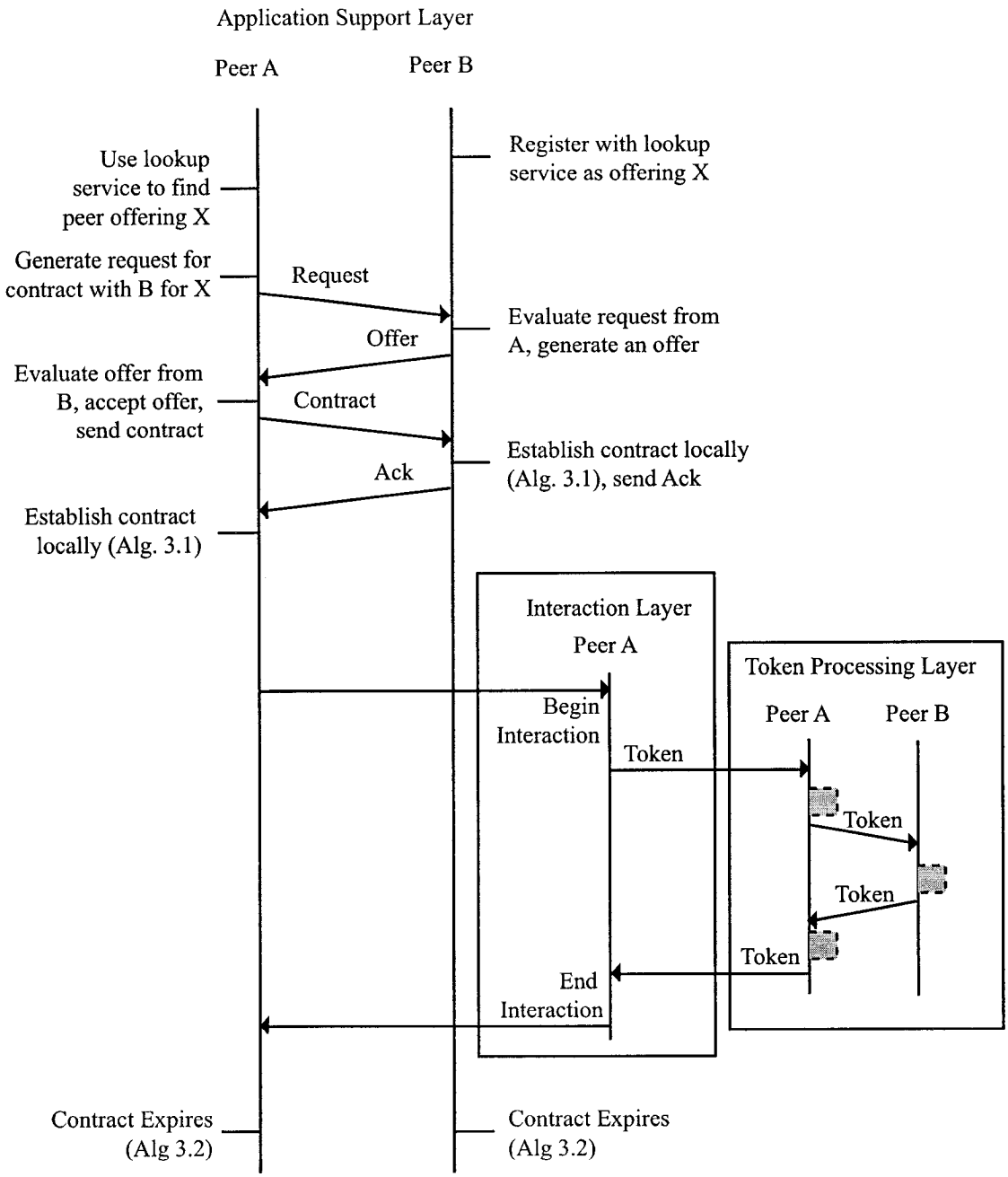

Figure 3.4: An illustration of the token-based peer-to-peer interaction coordination model. The shaded regions in the figure depict the period of time during which resources are held at a local site. The horizontal arrows in the figure represent transfer of control between local layers.

compensated since the amount of the original withdrawal is still in a container on the token if the microtransaction at B has failed.

3.3 Medical records example

To further illustrate the properties of TPIC interactions, consider an example in which a patient obtains a prescription from the doctor, fills it at their pharmacy, and has payment billed to their insurance company. In this scenario, pair-wise contracts exist between the patient and each of the other participants (doctor, pharmacy, and insurance agency), as well as contracts between the doctor and the pharmacy and between the pharmacy and the insurance company (see Figure 3.6).

Assume that the patient is ill and requires a prescription. At the application support layer on the patient's site, an interaction plan is established that adheres to all of the contract terms in the contracts held between the patient and the other participants. Since no access to global naming is assumed, a contract must exist between two participants for the token to travel from one to the other. At the time when the patient establishes a contract with the doctor, the contract negotiation can ensure that the doctor has a contract with the patient's pharmacy. This allows the token to move directly from the doctor to the pharmacy. Similarly, negotiation of a contract between the patient and the pharmacy can ensure that the pharmacy has a contract with the patient's insurance company. A typical interaction plan involving one doctor, one pharmacy, and one insurance company constructed in the same way as the example in Section 3.2 is shown in Figure 3.7.

In this simple example, the token plan could include all of the paths in the interaction plan. A more complex interaction plan could be constructed which includes a second pharmacy, where the token is shipped to this second pharmacy if processing fails at the first pharmacy. The initial token plan would be unchanged, but a processing failure at the pharmacy site could cause the application support layer at the patient site to issue a new token plan to the interaction layer that continues processing

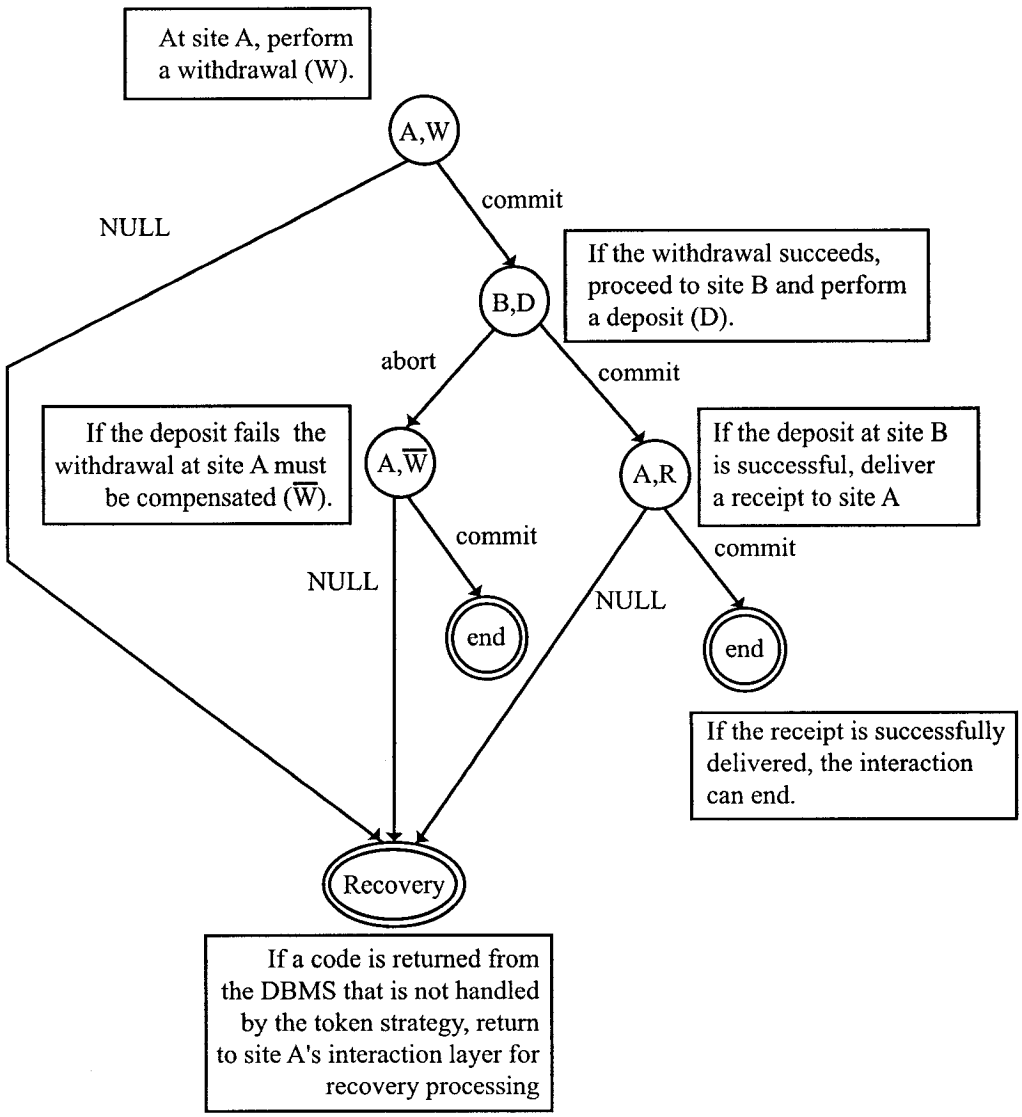

Figure 3.5: A token plan for the bank transfer example that includes a compensating microtransaction.

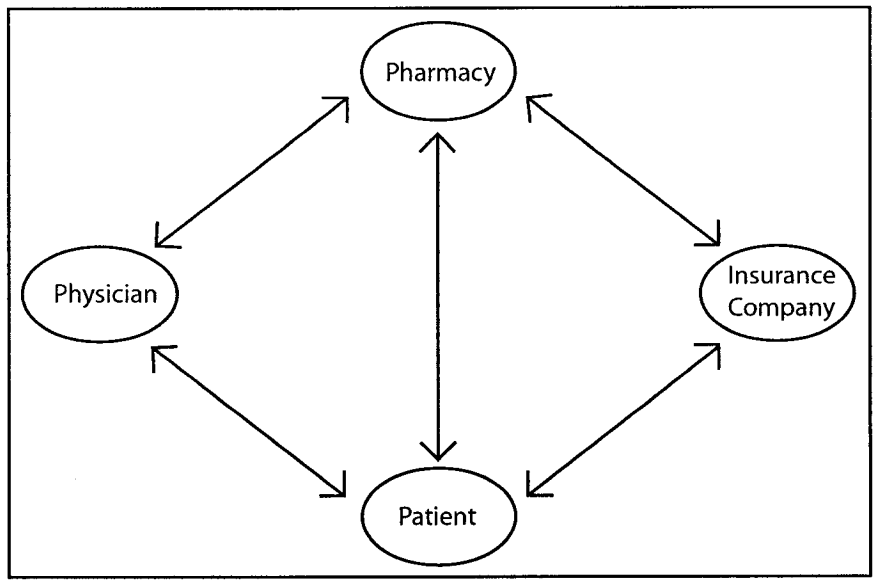

Figure 3.6: Contracts established between participants in the medical example.

using the other pharmacy.

In this example, the patient's medical history and prescription drug history reside under the patient's control. The medical history and prescription drug history are encrypted and placed in different containers on the token. When the token arrives at the doctor, the container with the patient's medical history is decrypted using a key in the patient-doctor contract. The doctor can then update the patient's medical history and encrypt it again before placing it back in a container on the token. The updated medical history is eventually returned to the patient's peer, where it will again be stored under the patient's control. The same applies to the patient's prescription history, which can only be decrypted by the pharmacy and the patient. This example shows how the TPIC model can be used to preserve end-user privacy while still allowing end-user participation in distributed transactions.

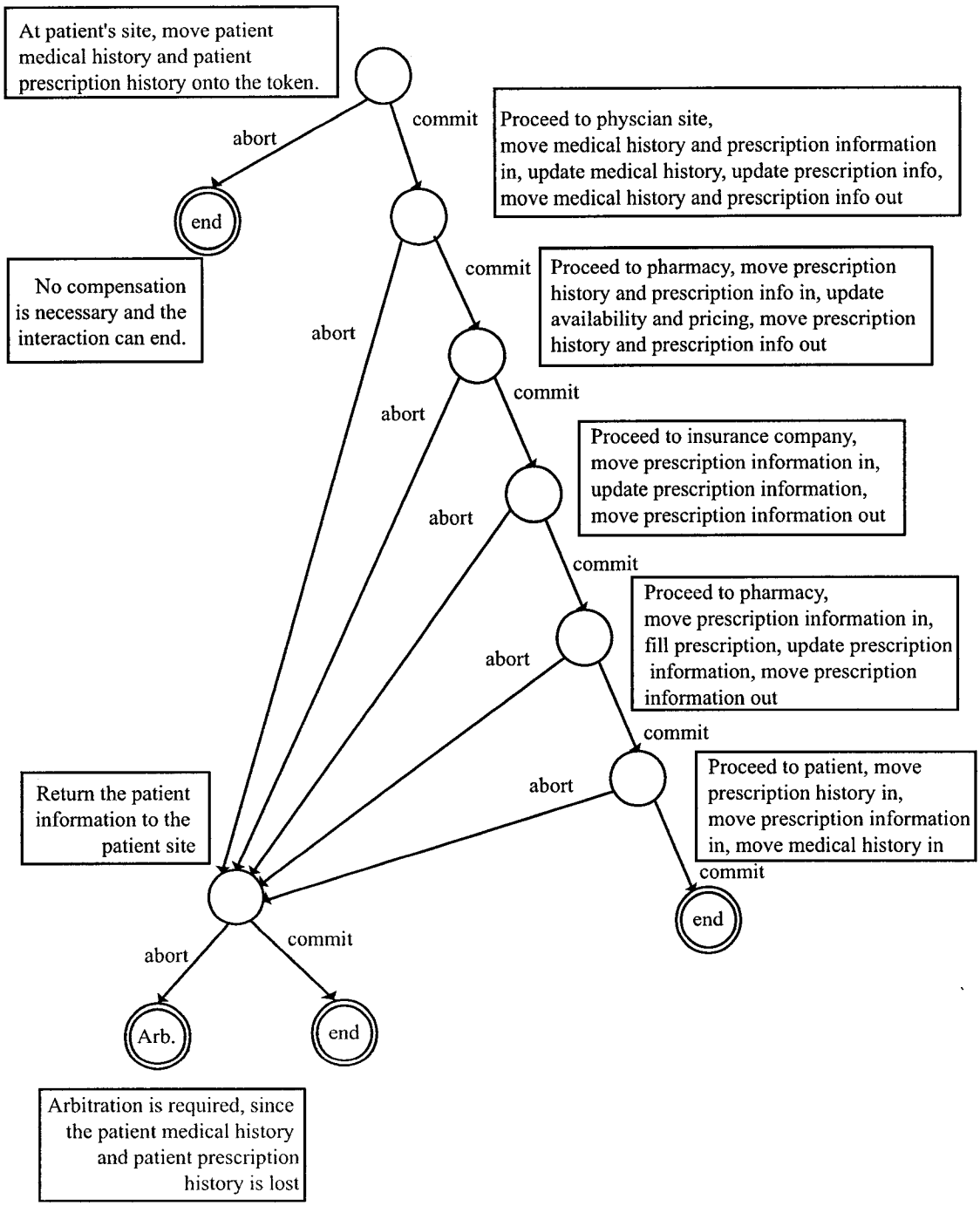

Figure 3.7: Interaction plan for the medical example.

Chapter 4

Detailed description of TPIC model

In this chapter, the TPIC model is described in detail by examining the roles of each layer of the model.

4.1 Contract layer

The TPIC model assumes that contracts are established before peer-to-peer interactions are initiated. Contract negotiation can be accomplished using an approach similar to the Contract-Net protocol [38] described in Section 2.10. Figure 4.1 shows the flow of messages during contract negotiation, establishment, and termination.

Domain experts establish the business tasks to be performed during a contract. Each business task maps to a microtransaction to be executed at a peer. Business rules specify the ordering and atomicity constraints across business tasks. For example, the policy "no delivery without payment", where delivery and payment are business tasks, can be expressed as the business rule (delivery $\Rightarrow$ payment). Each business policy can be expressed as a set of business rules involving business tasks. There is a direct mapping from business tasks to microtransactions, and a direct mapping from business rules to contract terms.

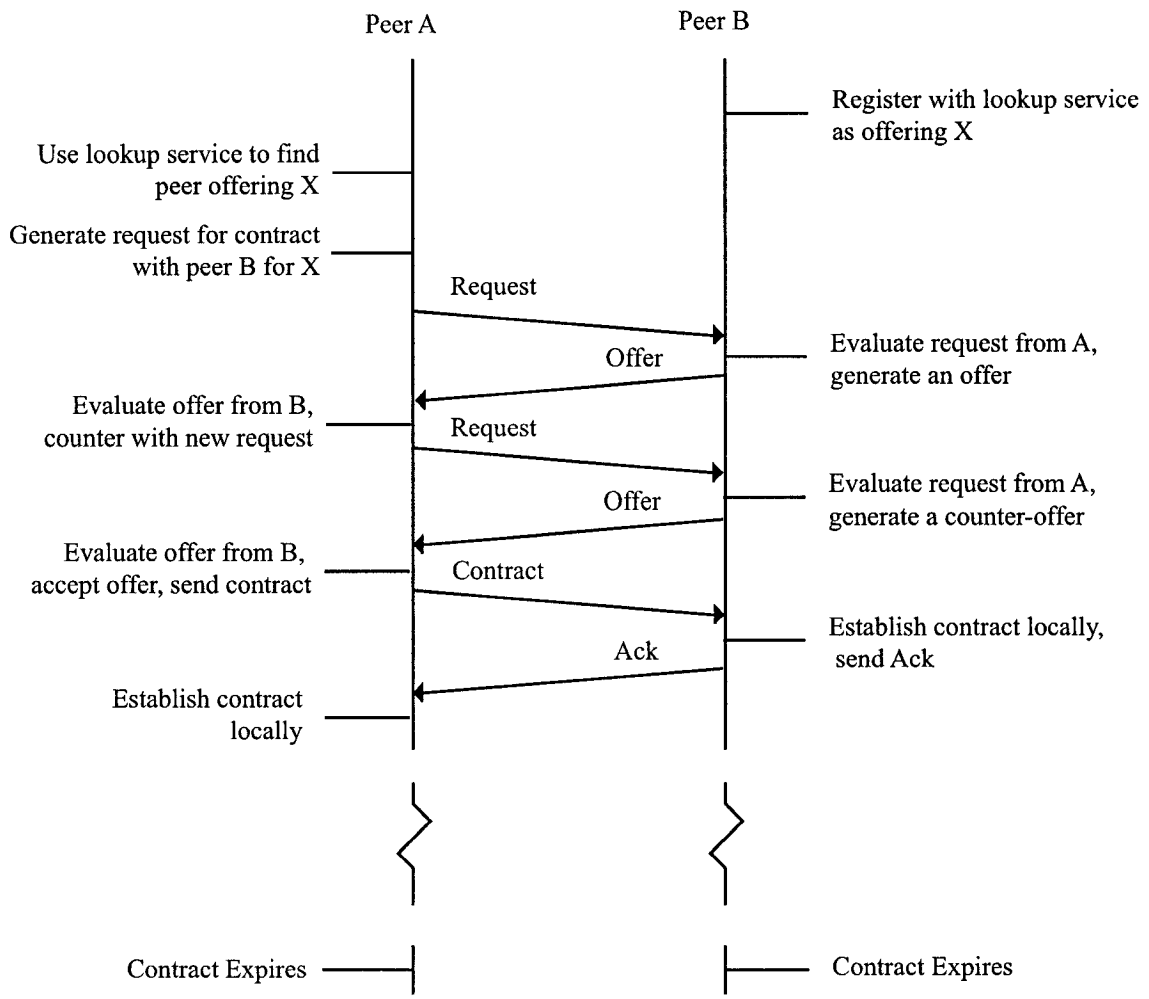

Figure 4.1: Timeline showing the events in contract negotiation, establishment, and termination

Definition 4.1 A *contract* between two peers P_i and P_j, denoted C_{ij}, is an ordered tuple (f, X, Y, Z, a, b, c) where f is a contract ID, X is a set of table descriptions, Y is a set of terms, Z is a set of references to SQL programs, a is an encryption key, b is a list of cancelled tokenIDs and c is a list of lost tokenIDs.

Definition 4.2 A *ContractID* is a unique identifier consisting of a concatenation of the HostIDs of the two hosts involved in the contract and a random digit string.

Definition 4.3 The *table descriptions* in the contract identify the temporary tables to be created in the local database at each of the peers involved in the contract, and also define tables in token containers.

The symmetry of the tables in the token containers and the temporary tables at each of the peers simplifies the process of moving data from a database into a token container and from a token container into a database. When a contract is established, these temporary tables are created in the databases of each of the peers. This creates catalog entries in each peer's DBMS catalog. When a contract expires, these temporary tables are deleted and the catalog entries are removed.

Definition 4.4 A contract *term* is an ordered pair (b_1, b_2) where b_1 and b_2 are micro-transactions that correspond to business tasks. Evaluation of a contract term results in a value of either true or false using the function $f(b_1, b_2) = (e_1 \Rightarrow e_2)$, where e_1 is true iff b_1 has been executed and has not been compensated, and e_2 is true iff b_2 has been executed and has not been compensated.

Definition 4.5 An *SQL reference* is a catalog reference to a parameterized SQL program in the catalog of the peer where execution will occur. The parameters are generated by the application support layer to suit the requirements of each interaction operating under a contract.

Definition 4.6 A *cancelled token list* is a list of tokenIDs that have been cancelled by the originating peer. This list is initially empty when a contract is established.

Definition 4.7 A *lost token list* is a list of tokenIDs that have been lost. The list is initially empty when a contract is established.

The *encryption key* in a contract is used to encrypt and decrypt containers so that the contents of containers are only accessible to the two peers participating in the contract.

The terms of the contract are static - to change them, a new contract must be negotiated and established. These terms provide a formal representation of the business rules governing the contract, which are used by the application support layer to produce an interaction plan. The SQL references are supplied by the peer offering service, and have parameters specified by the application support layer during token construction. While the contract terms are static and hold for all interactions, the SQL program to execute can be invoked with different parameter values for each interaction.

If an interaction cannot fulfill the terms of all of the contracts governing the interaction, arbitration will be requested. The contract layer is responsible for forwarding the details of the failed interaction and the relevant contracts to an arbitration process, which may be manual or automatic. Dealing with arbitration is outside the scope of this thesis. While arbitration is important for the correct execution of business transactions, it is less important with respect to each peer's database. The database at each peer will be in a consistent state before, during, and after the arbitration process occurs, since each microtransaction execution forms a complete database transaction.

4.2 Application support layer

The application support layer uses contracts that have been established by the contract layer along with domain-specific information and personalization information to produce an interaction plan and to construct token plans.

An interaction plan includes valid execution plans for an interaction, where a valid execution plan is one that satisfies all of the terms in each of the contracts involved

in the interaction. An interaction plan can be represented as a directed acyclic graph (DAG). The nodes in the graph correspond to microtransactions to be executed at specific peers, and the paths through the graph dictate the order in which peers will be visited. The choice among alternative successor nodes is based upon the code returned from the peer's DBMS during local processing (either "commit" or "abort").

The application support layer uses the services offered by the interaction layer to initiate peer-to-peer interactions. The TPIC model executes peer-to-peer interactions in a serialized fashion, where a token is moved from peer to peer and a microtransaction is executed at each peer. The information in the contracts includes references to SQL programs (and potentially their corresponding compensating SQL programs) to be executed at each peer. Additionally, constraints are supplied by the end-user through the application layer. This information is used by the application support layer to construct a plan detailing the order in which peers will be visited and the actions to be performed upon each visit, and to manage recovery processing if necessary.

Definition 4.8 A *link*, denoted r_{ij}, connects microtransaction s_i to microtransaction s_j, and is labeled with a return code v. Thus, the existence of the link $r_{i,j}$ with condition label v implies that s_j is the next microtransaction to execute if v is the return code from the execution of the SQL program referenced by s_i. The return code v will be either "commit" or "abort".

Definition 4.9 A *plan*, denoted G, is a directed acyclic graph with microtransactions for nodes and links for edges.

Definition 4.10 An *interaction plan*, denoted G_I, is a plan with a single source node and one or more sink nodes, wherein the hostID of every sink node matches that of the source node. Furthermore, the execution of any complete path from source to sink adheres to the contract terms of all of the contracts governing the interaction, or results in an escalation to arbitration. An interaction plan has a finite number of nodes.

Definition 4.11 A *token plan*, denoted H, is a DAG formed by taking a subgraph G_S of an interaction plan G_I with the property that all of the sink nodes of G_S are also sink nodes of G_I, and augmenting this subgraph by adding a special sink node s_e for handling exceptions and by adding links from every interior node to s_e with a NULL condition label. The NULL condition is produced when a return value is encountered that does not match the label of any outgoing link from the current node.

Definition 4.12 Two peers are in a *contractually conformant state* with respect to an interaction if all of the terms of the contract between the two peers governing the interaction are satisfied by the set of microtransactions executed by the interaction.

The terms governing a peer-to-peer interaction can be expressed as constraints on the execution of microtransactions in the interaction plan. A set of microtransactions can be bound together, such that all or none of the microtransactions must be completed. A microtransaction can belong to any number of binding sets. If there are no binding sets, then all of the microtransactions in the token plan are optional, and no compensation mechanism is required. Placing all of the microtransactions in a peer-to-peer interaction in a single binding set yields processing similar to Sagas [20], where any failure would automatically trigger compensating microtransactions for each of the microtransactions that have executed successfully.

Definition 4.13 An *InteractionID* is a three-part ID formed through the concatenation of the HostID of the originating peer, the ApplicationID of the peer-to-peer application that initiates the interaction, and a random digit-string.

Definition 4.14 A *Binding*, denoted B, is a set of microtransactions. An interaction plan must ensure that upon termination, either all of the microtransactions in B have committed, none of the microtransactions in B have committed, or the microtransactions in B that have committed have been compensated.

Definition 4.15 A *Partial Binding*, denoted Φ, is an ordered pair (J,K), where J and K are sets of microtransactions. An interaction plan must ensure that upon

termination, if any one of the microtransactions in K has not committed, then all of the microtransactions in J must not have committed or those microtransactions in J that have committed have been compensated.

The compensating microtransactions made available by the contracts can be used to reverse the effects of microtransactions that have completed. One approach to augmenting the interaction plan is to automatically compensate all of the microtransactions that have succeeded if one microtransaction fails. Thus, for any sequence of microtransactions $(m_1, m_2, ...m_i)$ that have successfully executed, the "abort" path for node m_{i+1} would lead to a sequence of compensating microtransactions $(\overline{m_i}, \overline{m_{i-1}}, ..., \overline{m_2}, \overline{m_1})$. This recovery mechanism is similar in form to that used in Sagas (see Section 2.3.3).

By making use of the bindings and partial bindings, it is possible to construct a large number of recovery paths to deal with any given failure. For example, if a microtransaction fails and no previously executed microtransactions are in bindings or partial bindings with that microtransaction, then the interaction may be able to proceed without compensating any microtransactions. It is possible to construct an interaction plan that will compensate as few microtransactions as possible and continue with the parts of the interaction that can proceed in the presence of a microtransaction failure. The middle ground between compensating everything and compensating as few microtransactions as possible is an area that can benefit from domain-specific optimization tools and heuristics.

Including all possible recovery paths in a token plan would lead to very large token plans. Rather than introduce the recovery paths on the token plan, the interaction plan can be extended to generate an appropriate recovery plan in the event of a failure. If a return value is produced that is not addressed in the token plan, execution of the token will cease and the token will be returned to the originating peer for recovery processing. At that point, the interaction plan can be extended to accommodate the specific failure that has been encountered and a recovery token plan can be constructed.

A token plan can be as simple as a single path from source to sink (plus the exception sink and related links). The exception sink is used to redirect the token back to the originating peer for recovery. The order in which the microtransactions are performed is dependent on domain-specific constraints and end-user preferences supplied by the application layer and the contract terms.

If token processing succeeds or fails for a given microtransaction, the token's plan will dictate where the token should travel next. If a failure occurs that is not handled by the token's plan, the default action is to return the token to the originating peer for recovery processing.

In the TPIC model, the local databases at which microtransactions have already executed are left in a consistent state. If a microtransaction fails, the token could be sent back to the originating peer with an error indication, at which point the application support layer can decide what corrective action to take. The fact that no peer's database consistency has been compromised and no locks are held by the interrupted interaction at the peer databases allows for a great deal of flexibility in choosing a recovery mechanism based on the circumstances of failure.

If a token plan completes successfully, the peer-to-peer interaction will terminate in a state that is consistent with the terms of the contracts governing the interaction. A contract might dictate that both the payment and shipment microtransaction must be completed, or neither one should be completed. This would be encoded by the two terms (delivery $\Rightarrow$ payment), (payment $\Rightarrow$ delivery). Any path through the interaction plan would then have to include either both microtransactions or neither microtransaction, and any recovery path in the peer-to-peer interaction that is between the two microtransactions would have to include a reversal of the earlier microtransaction. Once both of these microtransactions have completed, failure of the overall transaction may not require that they be reversed. If no other terms in the contracts place constraints on these two microtransactions, then failure of other parts of the peer-to-peer interaction may not require that these two microtransactions be compensated.

4.3 Interaction layer

The interaction layer processes requests from the application support layer and manages peer-to-peer interaction execution. The interaction layer generates tokens using the parameters passed in from the application support layer. A token includes a token plan dictating the path the token should follow under specific circumstances, a set of containers used to move data between peers, a token catalog, and a token log. The interaction layer delivers the token to the token processing layer where each microtransaction is executed in sequence at each of the peers detailed in the token plan.

The interaction layer maintains an interaction log, which includes an "initiate interaction" record for each interaction and a "terminate interaction" record for each interaction that has completed. Additionally, the interaction log has a copy of the token plan for each token that is issued, as well as a copy of the token log for each token that is returned. The purpose of the interaction log is to ensure that the interaction state can be preserved even if the initiating peer experiences a failure.

A token is used to execute microtransactions at peers in a specific order. The microtransactions to be executed together with the order in which the peers will be visited form the token plan. Containers on the token are used as volatile storage areas for moving data among peers. A log is kept on the token detailing where the token has traveled and the return code produced at each visited peer.

Definition 4.16 A *Token* is a tuple (z, G, K, L, U, c, s, e), where z is a TokenID, G is a token plan, K is a set of containers, L is a token log, U is a token catalog, c is a token counter, s is a status, and e is an error code.

Definition 4.17 A *TokenID* is a unique two-part ID consisting of the concatenation of the InteractionID of the interaction at the originating peer under which the token is executing and a random digit string.

Definition 4.18 A token *counter*, denoted c, is used as a relative timer that indicates the amount of time remaining on the originating peer's interaction timer. This can be

used to facilitate prioritization of token processing. The counter is only an estimate of the time remaining on the originating peer's interaction timer.

Definition 4.19 A token *status*, denoted s, is drawn from the set $\{Ready, Failed\}$.

Definition 4.20 A token *error code*, denoted e, is a code which is set when a token status is set to failed, and is used to pass information about the type of failure to the application support layer for help in recovery processing. Examples of error codes include *cancelled, no contract,* and *contract expired.*

Definition 4.21 A *Container* associated with a contract with ContractID f, denoted K_f, is an ordered pair (Q, R) where Q is a set of relational tables and R is a list of references to SQL programs.

Definition 4.22 A *Token Catalog* is a set of tuples (a, b, c, d) where a is a contract ID, b is a table schema, c is an access control list, and d is a row-count (set to 0 for empty table, -1 for uninitialized table). The contractID together with the table schema information uniquely identifies a table within a container on the token.

Definition 4.23 A *Token Log* is a set of ordered tuples (s, r, t) where s is a HostID, r is the return code from the microtransaction executed at peer s, and t is the value of the token counter at the time the log entry is written.

The purpose of the token log is to indicate which microtransactions in the token's plan have been attempted and the return code that was produced by this attempt. Given this information, the application support layer can determine the path through the interaction plan that was taken in order to produce a new plan in the event that recovery is invoked.

4.3.1 Failure-free processing

To simplify the description of token processing within the TPIC model, we begin with the case where token processing completes successfully within acceptable time

bounds and is free from failures. We then extend the interaction layer's functionality to accommodate recovery from failed microtransactions, cancellations, timeouts, and lost tokens.

Assuming that contracts have already been established an interaction plan has been generated, the application support layer initiates a peer-to-peer interaction. The interaction layer generates a token and submits the token to the token processing layer of the local peer.

Under failure–free processing, a single token is used to complete an entire peer-to-peer interaction. The token processing layer at each peer is responsible for executing a microtransaction at the local peer and forwarding the token to its next destination based on the token plan. The last microtransaction in a token plan will always be executed at the originating peer. Upon successful completion of this microtransaction, the token processing layer will pass the completed token up to the interaction processing layer.

When the interaction layer receives the completed token, the token's log is stored in the interaction log and the interaction terminates. A finite state diagram detailing the actions of the interaction layer under failure–free processing is shown in Figure 4.2. The figure shows the overall interaction layer processing finite state diagram with the components required for failure-free processing drawn using solid lines. Parts of the figure with dashed lines will be highlighted as they are described in later sections. Initiate Interaction is called by the application support layer when an interaction begins. The timer parameters and the steps assigning values to timers are not necessary for failure-free processing, but their role will become clear as the interaction processing mechanisms for dealing with failures are introduced in later sections. Terminate Interaction is called when a token is submitted to the interaction processing layer with a status of "Completed".

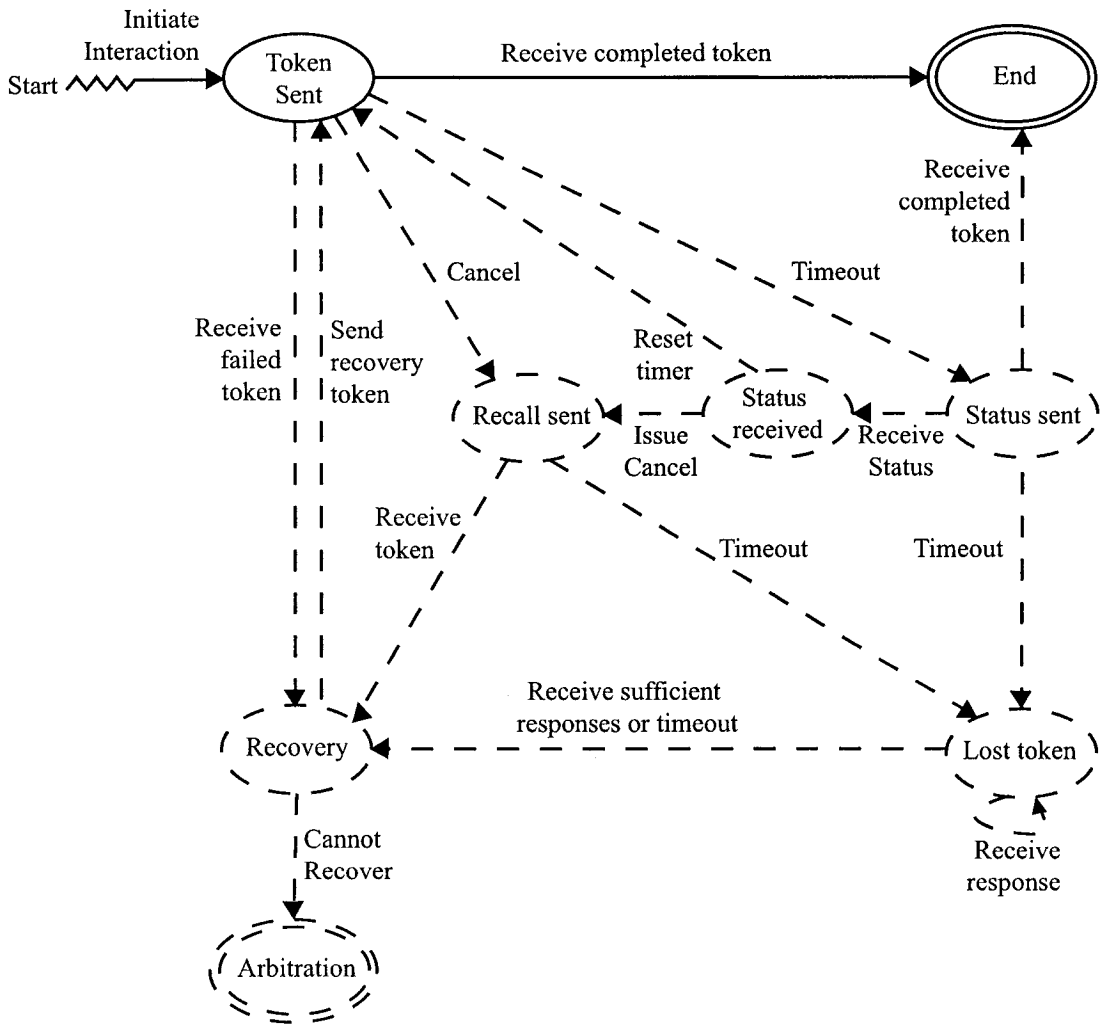

Figure 4.2: Finite state diagram for failure-free interaction processing.

4.3.2 Handling microtransaction failures

In the description of local token processing above, it is assumed that all microtransactions complete successfully. However, there are conditions under which microtransactions could fail. If a microtransaction fails for any reason, the token processing layer will set the status of the token to "Failed" and the token will be returned to the interaction processing layer of the initiating peer.

To handle recovery processing, new states are added to the finite state diagram of the interaction layer resulting in Figure 4.3. The arrival of a failed token causes a state change to the new "Recovery" state. This causes the interaction processing layer to request a new token plan from the application support layer. The application support layer will use the information in the token log to determine at what point in the overall interaction plan the token has failed, and will generate a new token plan that follows from that point to one or more sink nodes in the interaction plan.

If no recovery is possible, the application support layer will direct the interaction layer to move to "Arbitration". Recovery processing and arbitration may be required for reasons other than microtransaction failure, as detailed in the following subsections. Recovery is therefore discussed separately in a Section 4.4.

4.3.3 Handling Cancellations

While a token is moving among the peers involved in an interaction and executing microtransactions, the originating application layer may choose to cancel the interaction. The receipt of a Cancel signal from the application layer is passed through the application support layer to the interaction layer, and causes the interaction layer to generate and broadcast a cancel message to all peers in the outstanding token plan. The cancel message is processed by the token processing layer, and causes the TokenID of the outstanding token to be added to the contract's cancelled tokens list. Local token processing checks this list every time a token is received. If a token whose TokenID is on the cancelled list is received, the token is immediately sent back to the originating peer with a status of "failed" and an error code of "cancelled". After

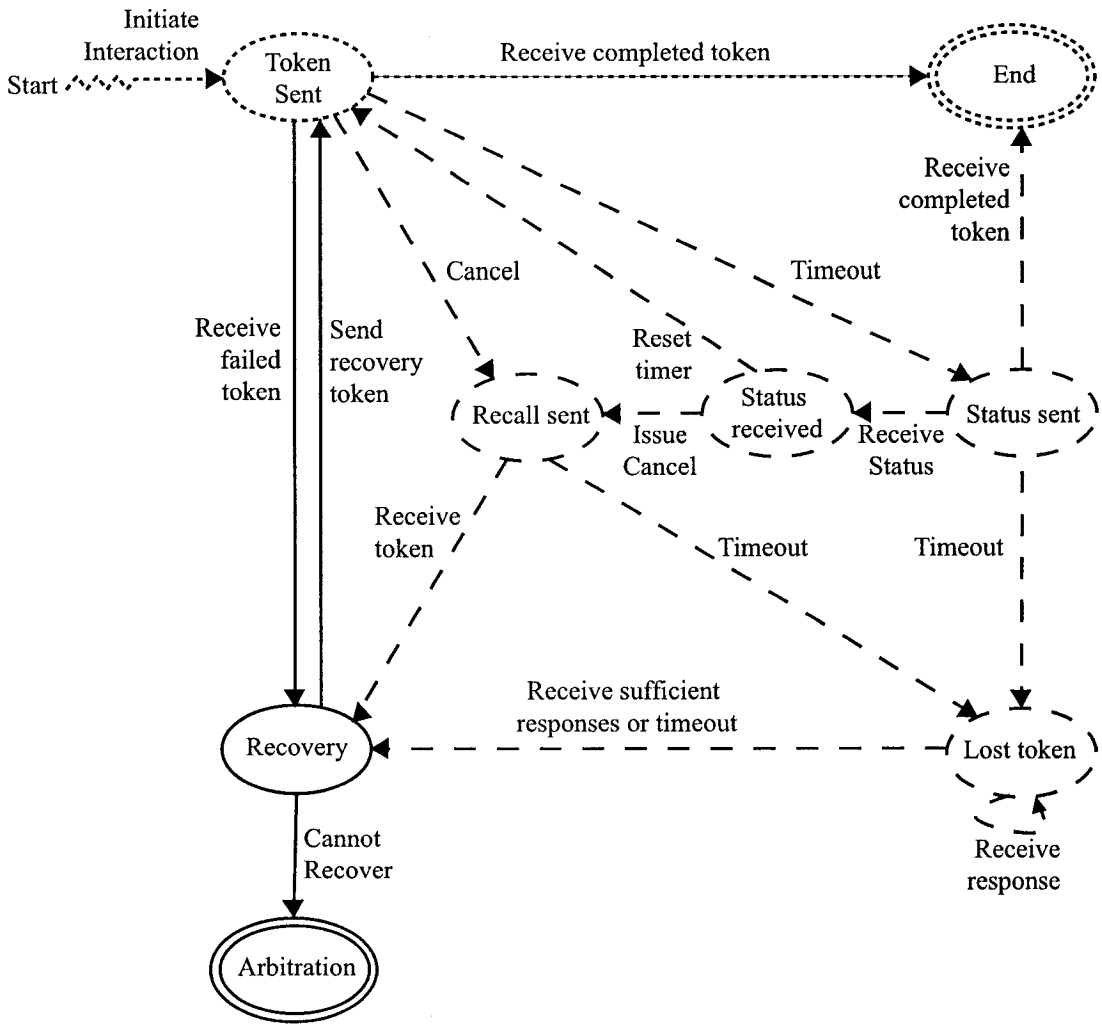

Figure 4.3: Finite state diagram for interaction processing in the presence of micro-transaction failures

a "cancel" message has been sent, the interaction layer expects to receive the token with a status of "Failed" and an error code of "Cancelled". After broadcasting a "Cancel" directive, the interaction layer waits for the failed token to be returned and sends it for recovery processing. This is broken out into a separate state in the figure to ensure that recovery is enacted even if the token returns as completed. At the token processing layer, the pre-processing step includes a check to see if the token that has arrived for processing is on the "Cancelled Tokens" list of the contract. Cancel Interaction is invoked at the interaction processing layer of the originating peer by the application support layer. The impact upon the interaction processing finite state diagram is shown in Figure 4.4.

4.3.4 Handling timeouts

The application support layer can specify a timeout parameter when initiating an interaction. The interaction layer uses this parameter to set a timer when a token is sent to the token processing layer. If the timer expires before the token returns, a status request is sent out to determine the microtransaction that the token has reached. Since the token's plan is a DAG and the choice of paths is decided on-the-fly, the originating peer does not know in advance where to send the status request. Therefore, the status request message is sent to the first peer in the outstanding token plan. If the interaction timer expires, Status Request will be executed at the interaction processing layer. The status request includes the TokenID of the outstanding token, and this TokenID is used to query the peer's DBMS log and determine where the token was sent.

When the status request message is received by the token processing layer of a peer, the TokenID in the request is checked against tokens in the local queues and against the TokenID of any currently executing token. If the token is found locally, a status response is sent back to the originating peer and delivered to the interaction layer. When a status response is received at the interaction processing layer, the interaction layer is moved to the state of Status Received. The impact on the interaction layer

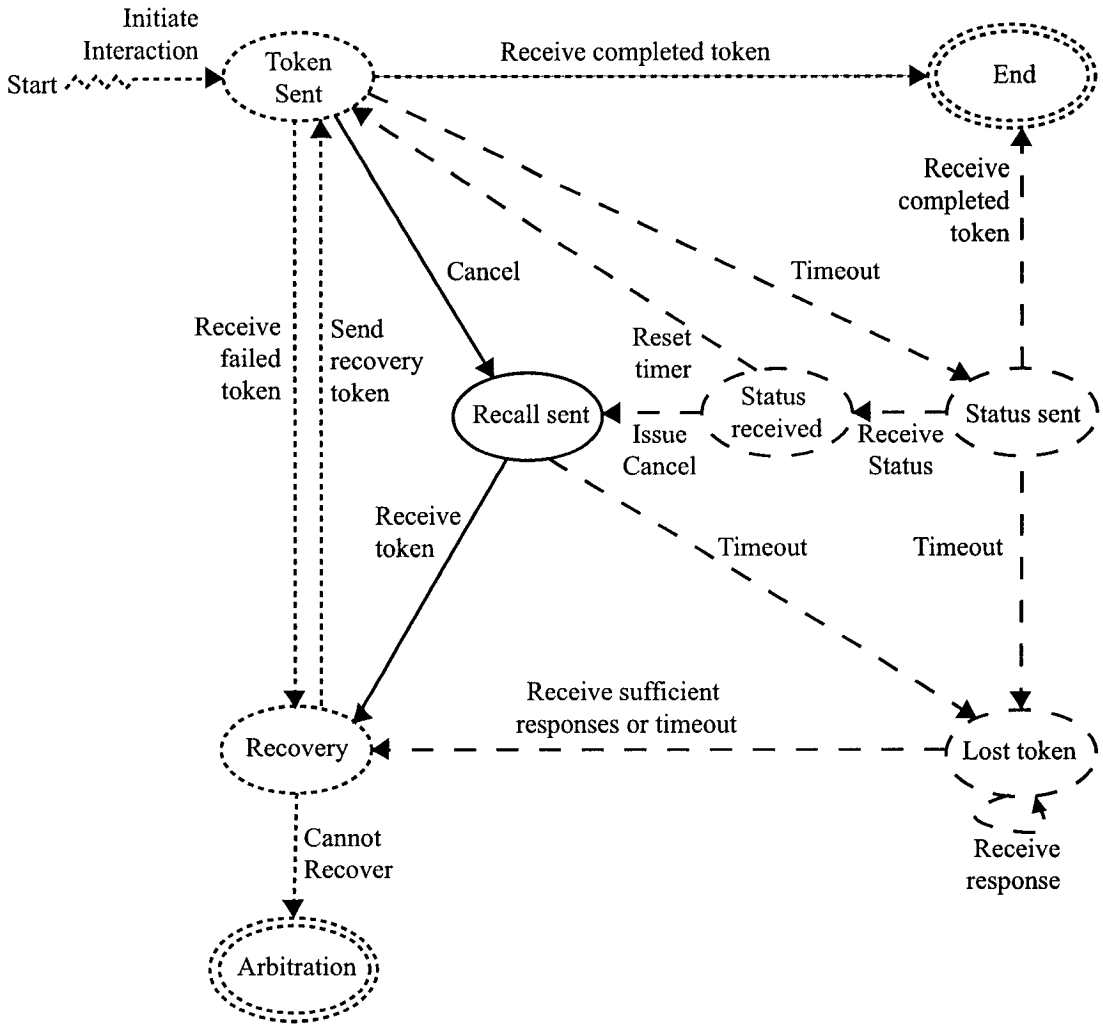

Figure 4.4: Finite state diagram for interaction processing with support for cancellations

finite state diagram is shown in Figure 4.5, which is either to cancel the token or reset the timer and continue processing.

4.3.5 Handling lost tokens

In order to handle lost tokens, the interaction layer of the originating peer must notice that an outstanding token has gone missing. This is accomplished through the use of timers at each of the states in the interaction layer where a token is outstanding. When a token is issued by the interaction layer, an interaction timer is set. When the timer expires, a status request message will be generated to determine the state of processing the token has reached, and the timer is reset to the status request timeout value. If the timer expires, it is assumed that both the token and the status request message have been lost, and lost token processing is initiated. If a status response is received before this timer expires, the application support layer is queried to determine whether processing should be cancelled or extra time should be given based on the stage that the token has reached. If processing is cancelled, a cancel message is broadcast to all peers listed in the outstanding token plan and the timer is set to the cancel timeout value. If the timer expires before the token arrives, it is assumed that the token is lost and lost token processing is initiated.

Information concerning the state of the token before it was lost is needed for recovery processing. A lost token query is broadcast to each peer listed in the token plan. When a lost token query is received at the token processing layer, the TokenID is added to the list of lost tokens in the contract. The peer's DBMS log and the contents of the stale temporary tables that were moved onto the token during processing are sent back to the interaction layer of the initiating peer. Based on the responses to the lost token query messages, the interaction layer may be able to determine at which peer the token was lost and may also be able to reconstruct the token as it existed before delivery to that peer. The reconstructed token can then be used for recovery purposes. If the lost token resurfaces, it will be discarded based on the presence of the TokenID on the list of lost tokens in the contracts of every peer in the

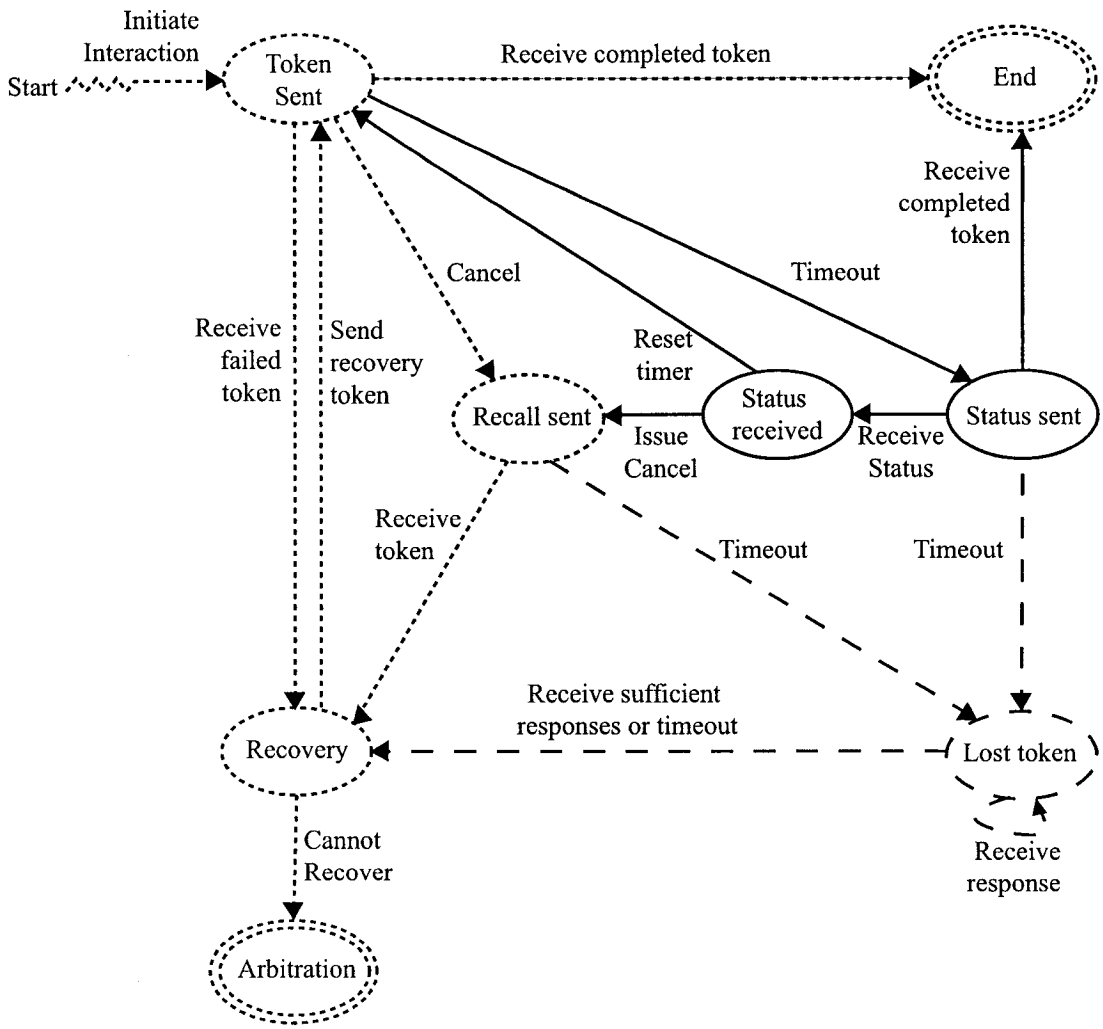

Figure 4.5: Finite state diagram for interaction processing with support for timeouts

token plan. If insufficient information is returned in response to the lost token query, arbitration may be necessary. Figure 4.6 shows the complete finite state diagram for the interaction processing layer with provisions for timeouts, cancellations, and lost tokens.

4.4 Recovery Processing

Recovery can occur at both the local processing level of a peer during the execution of a microtransaction, and at the interaction processing level. Recovery at the local processing level is handled entirely by the DBMS recovery processing facilities on the local peer. If token processing fails, recovery may be necessary at the interaction processing level because some of the microtransactions have committed and some have not. When the originating peer receives a failed token, the application support layer is sent the log of the token. The application support layer then determines how recovery should be effected. The application support layer uses the token's log to determine the microtransactions that were executed, and uses this information to determine at what point in the overall interaction plan the processing has reached. By default, the application support layer can generate a plan that will counter the effects of the original token's accomplishments in the reverse order that they were performed using compensating microtransactions. The application support layer is not, however, restricted to this choice of action. It may be determined that only some of the completed microtransactions need to be compensated, or that additional microtransactions can be performed that result in fulfillment of contractual obligations. Whatever the case, the application support layer will submit a new token plan to the interaction layer, and a new token will be generated and issued to continue processing. Any non-empty containers on the old token are transferred to the new token. If the application support layer cannot generate a new token plan, the interaction layer will move the interaction processing to arbitration and terminate the interaction.

In addition to handling processing failures due to failed microtransactions and

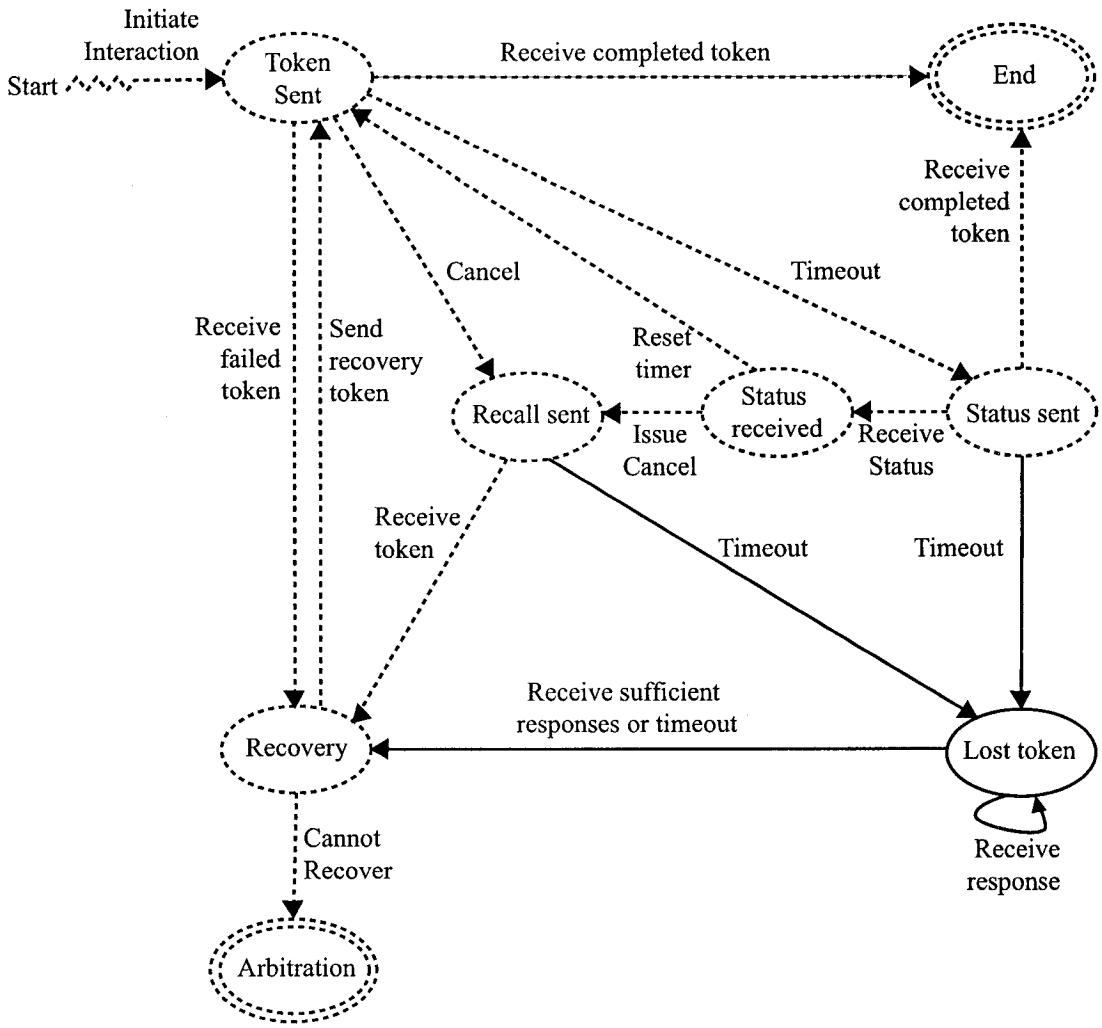

Figure 4.6: Finite state diagram for interaction processing with support for all types of token processing failure

lost tokens, the TPIC model must also handle failure of a peer participating in an interaction. There are four cases to consider: failure of a peer when the token is not present; failure of a node when the token is present; failure of the originating peer when the interaction has not yet completed; and failure of the originating peer after the interaction has completed.

4.4.1 Node failure - token not present

If a peer that is participating in an interaction fails when the token is not present, the peer's database will effectively recover from the failure. No special processing is required from the TPIC model, since any microtransaction that executed successfully at the failed peer would have completed before the failure occurred (since the token has already moved on), and any microtransaction that aborted at the failed peer prior to the peer failure would have been logged on the token as failed. The peer's DBMS recovery mechanisms are sufficient to handle this type of node failure.

4.4.2 Node failure - token present

In the case where a peer fails while a token is present at the peer, the token may be lost. Upon recovery, the stage of processing the token reached will be apparent in the peer's DBMS log. Since the token is lost, the lost token query will eventually be issued by the originating peer. The logs of the peer's DBMS will show the state of processing that was achieved and the temporary tables will have either the container values as they were when the microtransaction started (if the peer failed during execution of the microtransaction SQL program, or if the peer failed after aborting the execution of the SQL program) or the values as they were when the microtransaction completed successfully (if the peer failed after successfully completing the microtransaction but before delivering the token to the next peer in the plan). In either case, the contents of the peers DBMS log and the contents of the temporary tables are preserved by the peer's DBMS. Recovery can be effected based on the lost token processing.

4.4.3 Originating node failure - interaction ongoing

If the originating peer experiences a failure while an interaction is active, there will be a "initiate interaction" record in the interaction log with no corresponding "terminate interaction" record. The entries in the interaction log can be used to determine the state of the interaction plan that was reached by the last token that returned to the interaction processing layer. If the token was not present when the originating peer failed, it is possible that the token will arrive at the originating peer intact and complete, having continued processing the token plan while the originating peer was down. The information in the interaction log can be used to restore the interaction state at the originating peer.

If the token was present at the originating peer's token processing layer when the originating peer failed, the originating peer will eventually determine that the token was lost and initiate lost token processing as per Subsection 4.4.2. If the microtransaction completed successfully before the failure occurred, there will be an entry in the peer's DBMS log showing the completion of the microtransaction.

4.4.4 Originating node failure - interaction completed

If the originating peer experiences a failure after an interaction has completed, there will be a "initiate interaction" record and an "terminate interaction" record in the interaction log. Since the interaction had completed before the failure occurred, no special processing is required in support of the TPIC model. All of the microtransactions associated with the interaction were permanently committed to stable storage before the token was delivered to the interaction processing layer as completed, thus ensuring that the effects of the completed interaction will persist even in the presence of node failure.

4.5 Token processing layer

The token processing layer is responsible for executing microtransactions, processing cancel messages, processing status requests, and processing lost token requests. A finite state diagram showing the state transitions during token processing is shown in Figure 4.7.

Upon receiving a token, either from the interaction layer of the local peer or from the token processing layer of a remote peer, the token processing layer begins by moving processing to the pre-processing state. In this state, the lists of lost and cancelled tokens are checked against the token's ID, and the contract is checked to ensure it has not expired. If the TokenID is on the lost tokens list, the token is discarded and processing terminates. If the TokenID is on the cancelled token list or the contract has expired, the token's status is set to "Failed", the token is sent back to the originating peer and processing terminates. Otherwise, the token processing state is changes to "Moving data in'. In this state, containers on the token are decrypted using the encryption key in the contract and data is moved into temporary tables in the peer's database.

Once the data has been moved in, the SQL program associated with the current microtransaction of the token plan is submitted to the peer's DBMS and the token processing state is changed to "SQL Submitted". Upon completion of the SQL program, the processing state is moved to "Moving data out", and data is moved from the temporary database tables into containers on the token. A post-processing step encrypts the containers, adds an entry in the token's log, and determines if there are further microtransactions to be executed in the token's plan. If this was the last microtransaction, the token is handed up to the interaction layer on the local peer - otherwise, the token is forwarded to the token processing layer of the peer identified in the next microtransaction of the token plan.

The finite state diagram for handling cancel request messages at the token processing layer is shown in Figure 4.8. The cancel request message is broadcast by the

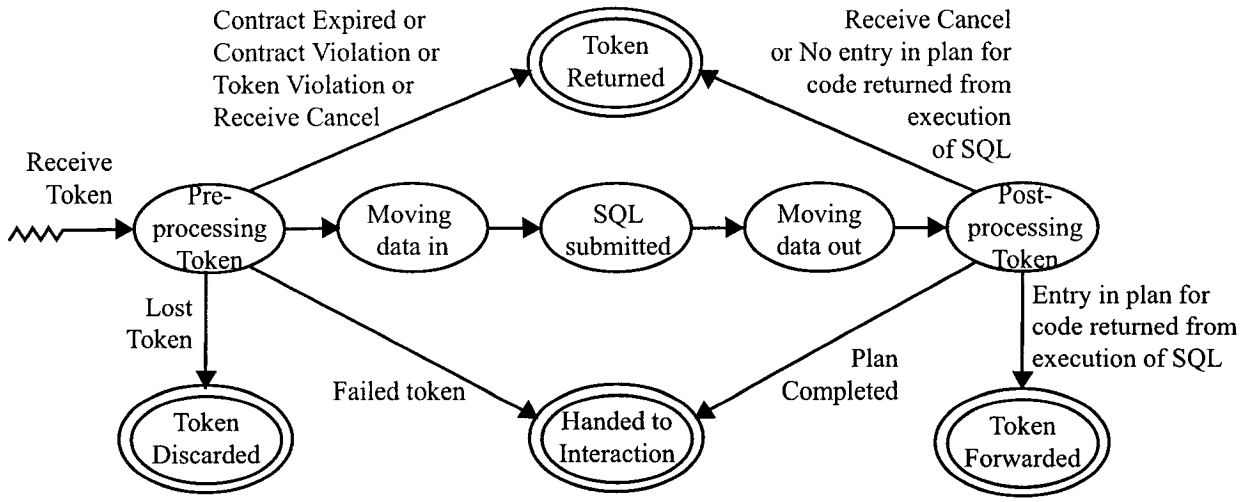

Figure 4.7: Finite state diagram for microtransaction processing

originating peer's interaction processing layer to all of the peers involved in an interaction. Each peer adds the TokenID of the cancelled token to the cancelled token list in its contract associated with the cancelled token's interaction.

Figure 4.9 displays the finite state diagram for handling status request messages at the token processing layer. A status request message is generated by the originating peer when the interaction timer expires before token processing completes. Upon receiving a status request message, a peer either forwards the status request along the path followed by the token, or delivers a status response message to the originating peer.

Figure 4.10 displays a finite state diagram detailing the actions performed at the token processing layer when a lost token message is received. A lost token message is broadcast to all of the peers involved in an interaction when the originating peer determines that the token has been lost. When a peer receives a lost token message, the TokenID of the lost token is added to the contract's list of lost tokens. The peer's database is queried to get a copy of the temporary tables associated with the lost token, as well as a copy of the local log entries based on the TokenID or a local mapping of the TokenID. All of this information is sent to the originating peer in the form of a lost token response message.

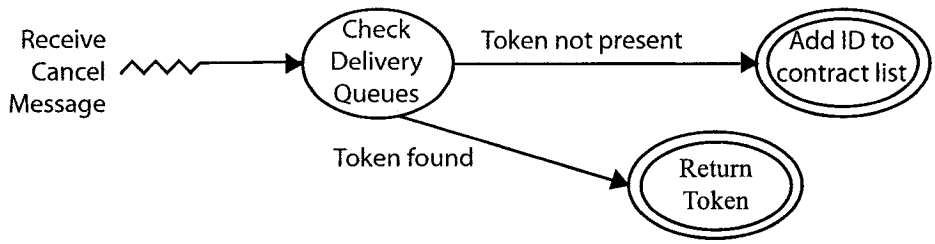

Figure 4.8: Finite state diagram for handling cancel request messages at the token processing layer

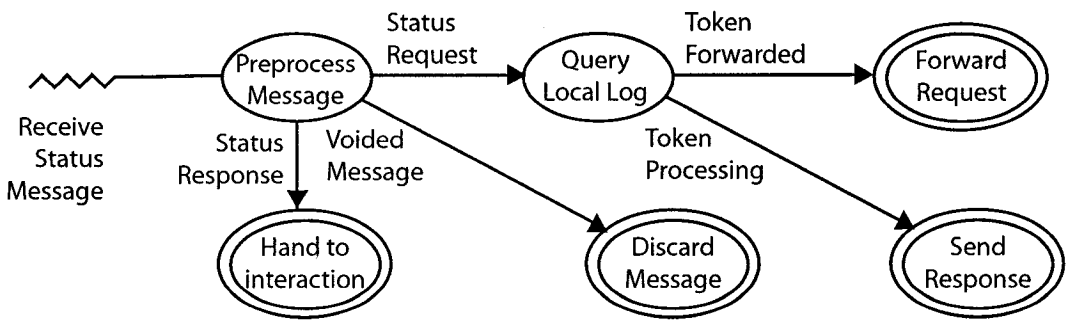

Figure 4.9: Finite state diagram for handling status request messages at the token processing layer

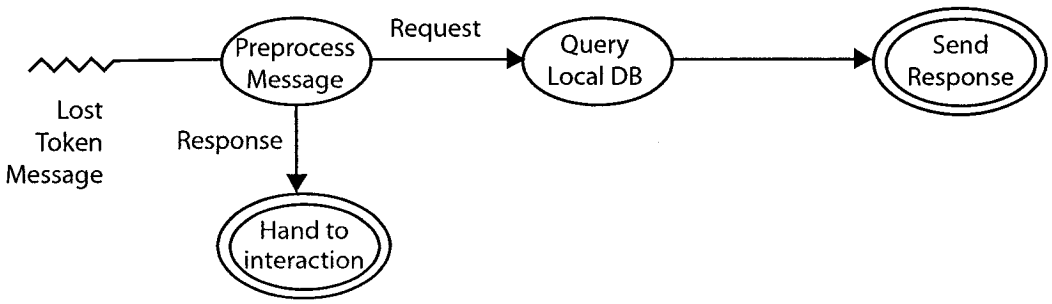

Figure 4.10: Finite state diagram for handling lost token messages at the token processing layer

Chapter 5

Token-based peer-to-peer interaction coordination

This chapter provides proofs of correctness for the execution of interactions under the TPIC model. This chapter begins with a description of the assumptions and design properties of the TPIC model. Next, it is shown that valid interaction plans can be generated that adhere to contract terms and domain-specific ordering constraints. Finally, it is shown that TPIC model interactions move each participating peer's DBMS from one consistent state to another consistent state, and also that TPIC interactions that are not escalated to arbitration move each participating peer from one contractually conformant state to another contractually conformant state with respect to the contract terms governing the interaction.

5.1 Assumptions

Assumptions 5.1 - 5.5 are fundamental assumptions about the environment in which the TPIC model will operate.

Assumption 5.1 Each peer involved in a peer-to-peer interaction maintains a DBMS.

Assumption 5.2 A peer's DBMS is a pre-existing database and its software cannot be modified to provide support for the TPIC model.

Assumption 5.3 Processing initiated by a remote peer under the model is viewed by the local peer's DBMS in the same was as processing initiated locally.

Assumptions 5.1-5.3 are similar to the assumptions underlying a multidatabase environment [2]. In contrast to an MDBS, there is no requirement in the TPIC model that the DBMSs support transaction processing. If there is support for transaction processing, remote requests will be submitted as complete local transactions. If the DBMS does not provide transaction processing support, remote requests will be submitted directly to the DBMS front end for scheduling.

Assumption 5.4 The mechanism in place for delivering tokens between peers is guaranteed to successfully deliver the entire token reliably or to not deliver the token at all.

Assumption 5.4 implies that tokens will not be lost in transit. This property could be eliminated by modelling communication failures as a failure of the receiving peer. Since discussion of communication failures would not add to an understanding of the TPIC model, this property of the underlying environment is assumed.

Assumption 5.5 Call-back functions from the interaction processing layer to the application support layer will complete in finite time.

Assumption 5.5 ensures that interaction processing will not be held up indefinitely when a decision is needed from the application support layer concerning how to proceed with the interaction.

5.2 Design properties

Design properties 5.1 - 5.8 describe characteristics of the TPIC model. These properties are important to bear in mind as they reflect model design decisions that have been taken.

Design property 5.1 The microtransactions in a peer-to-peer interaction are executed in a serialized fashion.

Design property 5.2 Each microtransaction in a peer-to-peer interaction forms a complete local transaction - that is, execution of a microtransaction either moves a local database forward from one consistent state to another consistent state or has no effect on the local database state.

Taken together, properties 5.1 and 5.2 imply that there can be at most one point of failure at a time in an interaction. Furthermore, a point of failure does not cause a local database inconsistency - all local databases are left in a consistent state. Thus, the recovery mechanism can be flexible and can operate under relaxed time constraints since local processing at each of the peers will not be blocked pending recovery.

Design property 5.3 Contracts are established between pairs of peers prior to their participation in TPIC model interactions.

This property rules out multi-party contracts. While not fundamental to the TPIC model, this property simplifies the discussion of the model and does not appear to limit its applicability. Multi-party contracts could be considered in the future as an extension to the TPIC model.

Design property 5.4 Temporary tables created in support of contracts are the same at both peers involved in the contract.

While it is not a requirement of the TPIC model, this property simplifies the transfer of data between tokens and peers. The data definitions for the temporary tables are established during contract negotiations. Allowing different temporary table definitions at each of the peers in a contract would require the introduction of a data mapping function during the transfer of data to and from a token's containers.

Design property 5.5 Processing of an SQL program by a peer's DBMS is guaranteed to complete in finite time, and will return a status code of either "commit" or "abort".

Existing relational DBMSs such as Oracle, Sybase, and DB2 adhere to design property 5.5.

Design property 5.6 Database constraints cannot be applied to the temporary tables used to move data on and off of token containers.

Design property 5.6 is required since the information in the temporary tables is stale once a microtransaction finishes executing.

Design Property 5.7 The TPIC model does not rely on global state.

Design Property 5.8 The TPIC model assumes only a single copy of data. A given data item is available to at most one peer at a time under this model.

Design properties 5.7 and 5.8 are introduced to support end-user privacy and end-user ownership of personal information, and force a model in which all local processing relies on locally available information. This leads to peer independence, allowing processing to continue at one peer regardless of the state of any other peer's processing.

5.3 Application Support Layer

A simple proofs shows that an interaction plan can be constructed for a given set of contract terms and ordering constraints.

Lemma 5.1 *Given a set of contract terms and a set of constraints that forms a partial order, there is an interaction plan whose execution paths adhere to the terms and constraints.*

Proof: The proof is by construction.

Arrange the microtransactions in a partial order, such that all of the ordering constraints are met. Create a total order of microtransactions by arbitrarily ordering microtransactions that are not comparable. Create an interaction plan with a node for each microtransaction, where the nodes are joined by links labelled with "commit" for the return code using the total order. For each node, and a link labelled with "abort" to escalation to arbitration.■

While this shows that an interaction plan can always be constructed, it results in arbitration any time a microtransaction aborts. This situation can be improved

by using compensating microtransactions to reverse the effects of those microtransactions that have successfully committed in the event that one of the microtransactions aborts. Thus, for any sequence of microtransactions $(m_1, m_2, ...m_i)$ that have successfully executed, the "abort" path for node m_{i+1} would lead to a sequence of compensating microtransactions $(\overline{m_i}, \overline{m_{i-1}}, ..., \overline{m_2}, \overline{m_1})$. Each new node added to the interaction plan would have an "abort" path that leads to arbitration, meaning that arbitration is needed when a compensating microtransaction fails. This recovery mechanism is similar in form to that used in Sagas (see Section 2.3.3).

While introducing compensation microtransactions should lead to less need for arbitration, an intelligent planning tool could take advantage of the constraints in order to further refine the interaction plan and support a finer granularity of atomicity among microtransactions. To this end, the following construction approach for generating sets of bindings and partial bindings can provide information to such a tool to allow for the construction of flexible interaction plans which still adheres to contract constraints.

For each ordering constraint (m_i, m_j), add an explicit term (m_j, m_i) to the set of contract terms Y. Given this augmented set of contract terms Y, if the set includes both (m_i, m_j) and (m_j, m_i) for some i, j, then m_i and m_j are in a binding $B = \{m_i, m_j\}$. If (m_i, m_j) is included in the set of terms but (m_j, m_i) is not included in the set, then m_i and m_j are in a partial binding $\Phi = \{\{m_i\}, \{m_j\}\}$.

Next, the number of bindings and partial bindings is reduced by generating bindings and partial bindings with more than two microtransactions. If two binding sets share a common microtransaction, then the union of the two binding sets is a binding set that can replace the two binding sets. Thus, given B_1, B_2 where $m_i \in (B_1 \bigcap B_2)$, $B_3 = B_1 \bigcup B_2$ is a binding set that replaces B_1 and B_2. Similarly, if two partial bindings Φ_1, Φ_2 share a common set, the two partial bindings can be combined. If $\Phi_1 = (M_i, M_j)$ and $\Phi_2 = (M_i, M_k)$, then $\Phi_3 = (M_i, M_j \bigcup M_k)$ is a partial binding set that replaces Φ_1 and Φ_2. If $\Phi_1 = (M_i, M_j)$ and $\Phi_2 = (M_k, M_j)$, then $\Phi_3 = (M_i \bigcup M_k, M_j)$ is a partial binding set that replaces Φ_1 and Φ_2.

Once the number of binding sets and partial binding sets has been reduced, the resulting sets are used to build the interaction plan. The recovery path from an aborted microtransaction need only include compensating microtransactions based on their membership in a binding or partial binding with the aborted microtransactions. The bindings and partial bindings add value is in the construction of interaction plans that can handle failures in a flexible manner, leading to a reduction in the number of circumstances that cause escalation of the interaction to arbitration.

5.4 Interaction and Token Processing Layers

Conceptually, a Database Management System (DBMS) consists of two major components, a front end pre-processor which accepts and parses SQL programs, and a back end data manager responsible for maintaining the stored database state [14]. In the TPIC model, each peer maintains an independent DBMS. An SQL program submitted to the front end is guaranteed to terminate and will move the database from one consistent state to another consistent state, either by storing all of the changes associated with the execution of the SQL program or by using the log to undo and changes that occur before a failure. When execution of an SQL program terminates, the front end will return a status code of either "commit" or "abort".

SQL programs can only be issued to the front end by the peer maintaining the DBMS. A single microtransaction's execution involves the invocation of a function on the local peer passing an SQL program for execution on the local database. The local peer is responsible for scheduling and execution of this request with the peer's DBMS. The peer's DBMS will complete the requested SQL program in finite time, resulting in either "abort" or "commit". If the SQL program is running for too long, it will be cancelled by the peer's DBMS based on a timeout.

Lemma 5.2 *Processing of a single microtransaction terminates.*

Proof: The processing of a single microtransaction is shown as a finite state diagram in Figure 4.7. The finite state diagram is a sequence of states with no explicit

loops. A state transition occurs when the processing related to a state terminates. Each state that involves assignment of a value or testing of a value requires constant time to execute. Each state that queues the token for delivery takes time proportional to the size of the token, and delivery is delegated to the reliable infrastructure (see Property 5.7). The states which involve checking set membership require time proportional to the size of the sets (the local sets that may be checked include the set of contracts, the set of cancelled tokens, and the set of lost tokens). Moving data between the container and the peer's DBMS will require time proportional to the amount of data being transferred. The execution of an SQL program is guaranteed to complete in finite time by Property 5.5. In all cases, the processing at each state will complete within finite time, therefore the microtransaction will complete in finite time. ■

Lemma 5.3 *Execution of a single microtransaction at a peer will move the peer's database from one consistent state to another consistent state*

Proof: Processing of a microtransaction at a peer is governed by the finite state diagram in Figure 4.7. Processing the only three states affects the peer's DBMS: the movement of data from the token into temporary database tables; the execution of the SQL program; and the movement of data from the temporary database tables into the token. Execution of the SQL program will result in a success or a failure, in which case either all of the changes associated with the execution are effected, or none of the changes are effected, respectively. Since the microtransaction's activities on the peer's DBMS are scheduled as a local SQL program, the peer's DBMS processing guarantees the preservation of local consistency (by Property 5.5). By Property 5.6, the data in the temporary tables cannot affect the consistency of the peer's database since these tables are not involved in any DBMS constraints. ■

Lemma 5.4 *Processing of a token plan terminates*

Proof: A token plan is as an acyclic, directed graph with microtransactions as nodes and links as edges (see Definition 4.11). Processing of a token plan involves

executing microtransactions and transitioning links based on the return codes of microtransaction executions. Processing of a token plan completes when the microtransaction referenced by a sink node of the token plan completes, or when a null transition link is followed back to the originating peer. Each microtransaction is guaranteed to terminate (by Lemma 5.2). The transition from one node to another involves traversing a link in the token plan, and moving the token from one peer to another in the environment. The movement of the token is guaranteed to complete successfully or to be held in a local queue awaiting transmission by Assumption 5.4. There are three conditions of interest: token movements are successful; the token is held indefinitely in a queue pending delivery; or the token is lost due to a node failure.

In the case that token movements among peers are successful, processing of the token plan will terminate.

In the case of a token being held in a queue indefinitely pending delivery to the next peer in the token plan, the interaction timer at the originating peer will eventually expire. Upon expiration, the interaction processing layer will cause a token status message to be propagated to the node with the token, and the originating peer will be informed of the token's status. Based on the status, the originating peer may extend the time for the token's execution by resetting the interaction timer, but this timer can only be reset a fixed number of times during a given interaction (the actual number of times the timer can be reset is a parameter passed to the interaction processing layer when the interaction is initiated). If the maximum number of timeouts occurs, the originating peer will issue a cancel message. Receipt of a cancel message at the token processing layer will cause the token to be cancelled and sent back to the originating peer, causing processing of the token plan to terminate.

In the case that the token is lost, expired timers at the originating peer will eventually detect the loss of the token. The originating peer then forces termination by broadcasting a lost token message to all peers involved in the interaction. Once a token's ID has been added to the list of lost tokenIDs at a peer, the token's plan will never be processed at that peer. If the token re-surfaces, it will be discarded without

further processing.

Thus, the processing of the token's plan terminates in all cases. ■

Lemma 5.5 *Processing of an interaction plan terminates.*

Proof: Processing of an interaction plan terminates when a "terminate interaction" record is placed in the interaction log. This can happen for one of two reasons: processing of the interaction plan has reached a sink node; or the interaction has been escalated to arbitration. Processing of an interaction plan is accomplished by the processing of one or more token plans derived from the interaction plan. Each time the interaction layer delivers a token to the token processing layer, the processing of the token plan by the token processing layer is guaranteed to terminate (by Lemma 5.4). If the token successfully completes processing its plan, the interaction terminates. If the token's processing is cancelled or fails, the interaction processing layer will execute a call-back routine to the application support layer requesting instruction. Each time the interaction layer requests information from the application support layer, the application support layer is guaranteed to respond to the interaction processing layer within finite time by Assumption 5.5. The originating peer may issue a new token that will retry, compensate, or pursue an alternative to the failed processing. By Definition 4.10, there will be a finite number of alternatives to attempt. When there are no further alternatives to attempt, the detailed logs of all tokens involved in the interaction are used to determine what should happen based on the contracts involved in the interaction through arbitration. Once the required information has been sent to arbitration, the interaction terminates. Thus, in all cases, processing of an interaction plan terminates. ■

Lemma 5.6 *The TPIC model preserves the durability property with respect to completed interactions.*

Proof: Durability is the property that the effects of completed interactions will persist even in the presence of failures. By Lemma 5.3, each microtransaction that completes at a peer is made durable by the peer's DBMS. In the event of a failure of the originating peer during processing, the interaction log is used to determine

if an interaction has successfully completed or was still in progress when the failure occurred. Any interactions with a "terminate interaction" record in the interaction log were completed before the failure occurred. The effects of the completed interaction were already made durable by the microtransactions executing at the peers involved in the interaction. The presence of the "terminate interaction" record implies that no further processing is required with respect to this interaction. Since the effects of the interaction are made durable by the DBMSs of the peers involved in the interaction, the durability of the effects of an interaction that has completed are guaranteed.

If an interaction has a "initiate interaction" record in the log, but there is no matching "terminate interaction" record, the interaction was still in progress when the failure occurred. The interaction can continue on the basis of the information in the interaction log coupled with the result of a status query. The effects of an interaction that has not completed before failure may have to be compensated, or the interaction may have to be sent to arbitration. The decision to continue processing or escalate to arbitration is based on the contents of the interaction log and the result of a status query to determine the status of the token. ■

Theorem 5.1 *Interactions are recoverable in the presence of node failures.*

Proof: From Lemma 5.3, the microtransactions that have successfully completed within an interaction are durable in the presence of node failures. As discussed in Section 4.4, each peer's DBMS recovery mechanisms will ensure that completed microtransactions are durable, and any microtransactions that were incomplete when a failure occurred have their effects reversed during local recovery processing. From the interaction processing perspective, the local recovery mechanisms at each peer involved in the interaction handle the recovery of the peer DBMSs. If the originating peer experiences a failure, recovery processing is required for interactions that have been initiated but have not yet terminated. The interaction log is used to determine if an interaction has terminated. If there is a "terminate interaction" record in the log, all of the effects of the interaction are guaranteed to have been made durable by Lemma 5.6. If there is an "initiate interaction" record in the log with no corresponding

"terminate interaction" record, the log is used to determine the state of processing the interaction achieved before the failure occurred. Any tokens issued by the interaction that completed before the failure occurred will have their logs in the interaction log. If a token was in progress when the failure occurred, the interaction log will have a copy of the token's plan. Using the logs of completed tokens and the plan of the outstanding token, the application support layer can reconstruct the interaction plan governing the interaction and determine the set of microtransactions that have been successfully completed by the interaction, thus allowing the interaction processing to proceed from the point of failure. ■

Lemma 5.7 *Two peers are in a contractually conformant state with respect to an interaction before the interaction begins.*

Proof: Before an interaction begins, no microtransactions have been executed by the interaction, thus the set of microtransactions that have been performed by the interaction with respect to the two peers is the empty set. This implies that all of the terms of the contract between the two peers are satisfied with respect to the interaction, and the two peers are in a contractually conformant state with respect to the interaction (by Definition 4.12). ■

Lemma 5.8 *Two peers are in a contractually conformant state with respect to an interaction after an interaction completes, assuming that the interaction was not escalated to arbitration.*

Proof: For an interaction to complete without having escalated to arbitration, the final token issued by the interaction has to complete successfully. The overall set of microtransactions executed by an interaction consists of the concatenation of one or more token plan histories. The initial token plan generated by an interaction must begin with the interaction's source node. Successful completion of a token plan will result in a token plan history whose last node is a sink node of the interaction plan. Thus, the concatenation of the token plan histories from a completed interaction that was not escalated to arbitration will form a path from the source node of

the interaction plan to one of the sink nodes of the interaction plan. The set of microtransactions executed by the interaction must therefore adhere to all of the terms of all of the contracts governing the interaction, by Definition 4.10, including the two peers in question. ■

Theorem 5.2 *Execution of a TPIC interaction that is not escalated to arbitration moves the peers involved in the interaction from one contractually conformant state to another contractually conformant state with respect to the interaction, and the microtransactions executed by the interaction move each peer's DBMS from one consistent state to another consistent state.*

Proof: To prove this theorem, it will be shown that each individual microtransaction executed by an interaction at a peer moves the peer's database from one consistent state to another consistent state, thus preserving database consistency at each of the peers, and it will also be shown that the pairwise changes to any two peers effected by an interaction conform to the contract between those two peers, yielding a model of interaction that is consistent with the terms of the contracts governing the interaction.

For each microtransaction executed by an interaction, the microtransaction moves the state of a peer DBMS from one consistent state to another consistent state by Lemma 5.3.

By Lemma 5.7, all of the peers involved in the interaction are in a contractually conformant state with respect to the interaction before the interaction begins.

By Lemma 5.5, processing of an interaction plan terminates. If this termination is not due to an escalation to arbitration, then the final token plan issued to the interaction layer completed successfully.

By Lemma 5.8, all of the peers involved in the interaction are in a contractually conformant state with respect to the interaction after the interaction terminates if the interaction was not escalated to arbitration. ■

Chapter 6

TPIC Model Implementation and Validation

This chapter describes algorithms which can be used to develop an implementation of the TPIC model. The algorithms involved in failure-free token processing (Algorithms 3.3, 3.4, and 3.10) have been validated through a preliminary implementation written in Perl running on a UNIX environment with access to an Oracle DBMS. In this preliminary implementation, layers of the TPIC model were developed as Perl scripts. Multiple sites were simulated on the same server. Each virtual site maintained its own connection to the Oracle DBMS, with access to its own set of tables. Message passing between layers and across virtual peer instances was accomplished by writing/reading files in common directories on the local disk.

A more complete implementation of the TPIC model could be used to test the model's correct performance under a variety of situations. Examining interaction logs and token logs in the presence of simulated node failures would serve to test the model's ability to preserve DBMS consistency and global contractual conformance in the presence of failures. In particular, the interaction layer and processing layer algorithms needed to properly move the interaction state and process the token plan could be tested by implementing a rudimentary prototype system with stubs inserted for contract establishment and interaction plan generation modules. This would allow

testing to ensure that given a valid token plan that adheres to established contracts, the TPIC model can successfully execute interactions and successfully recover from failures.

6.1 Implementation

The preliminary implementation has been completed for failure-free processing conditions (Algorithms 6.3, 6.4, and 6.10), and interaction processing for the medical example discussed in Section 3.3 has been successfully demonstrated. Interaction plan generation and token plan selection at the application support layer were hard-coded as opposed to dynamically generated using tools from AI.

In the TPIC model, if token processing does not complete successfully the recovery processing algorithm (Algorithm 6.9) submits the log of the token to the application support layer for a decision about what to do. The application support layer uses the log to determine the progress made through the interaction plan, and either produces a new token plan that can be used to successfully complete interaction processing or opts for arbitration. If arbitration is requested, the interaction processing layer moves to arbitration and terminates processing. Otherwise, the new plan submitted to the interaction processing layer by the application support layer is used to construct a new token, and processing of this new token operates exactly as in normal operation. Arbitration of failed interactions is outside of the scope of the system. Implementation of cancellation, status, and lost token processing will allow testing of recovery from node failures.

Algorithms 6.1, Establish_Contract, and 6.2, Terminate_Contract, show the steps involved in establishing and terminating contracts, respectively. The simulated peer-to-peer application layer establishes the required contracts and proceeds to request the application support layer to initiate an interaction.

Algorithm 6.1 Establish_Contract(RemoteHostID, Terms, TableDescriptions, EncryptionKey, SQLReferences)

Generate a unique ContractID using local HostID and remote HostID
Create temporary tables in each peer's DBMS from TableDescriptions
Store Terms, TableDescriptions, EncryptionKey, SQLReferences

Algorithm 6.2 Terminate_Contract(ContractID)

Delete temporary tables in each peer's DBMS from TableDescriptions

6.2 Interaction Layer

The algorithms described in this section describe the implementation of the interaction layer. Each algorithm changes the state of the interaction layer as shown in Figure 6.1.

Algorithm 6.3, Initiate_Interaction, is used to initialize interaction parameters, construct and delivery a token to the token processing layer, and set the interaction to the "Token Sent" state. Under normal processing circumstances, the token will return to the interaction processing layer with a token status of "Completed" and Algorithm 6.4, Terminate_Interaction, is used to log the token plan, return the results of the interaction to the application support layer, add a "terminate interaction" record to the interaction log, and move the state of the interaction to the "End" state.

Under failure free processing, only Algorithms 6.3 and 6.4 are required to implement the interaction layer. Algorithms 6.5 through 6.9 are required to handle cancellations, timeouts, status requests, lost tokens, and recovery processing at the interaction layer.

In the event that the application support layer issues a request to cancel an interaction, Algorithm 6.5, CancelInteraction, is invoked by the interaction processing layer. This algorithm causes a cancel message to be broadcast to all of the sites participating in the interaction, and effectively moves the interaction's state to the "Recall Sent" state.

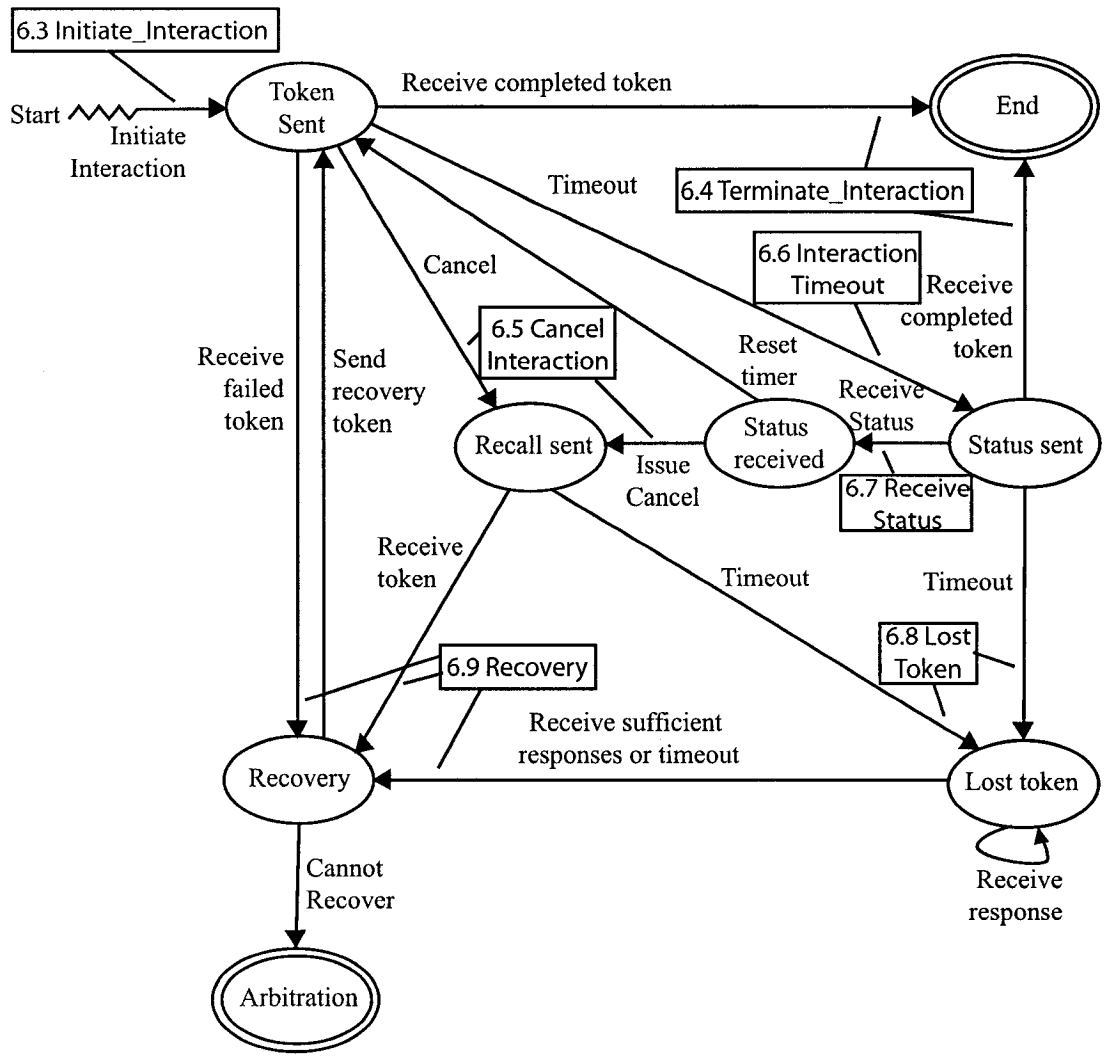

Figure 6.1: Interaction processing finite state diagram labelled with algorithms used to change from one state to another. The boxed elements in the figure are the algorithms.

Algorithm 6.3 Initiate_Interaction(AppInstID, Microtransactions, TokenPlan, SQL-References, TableSchemas, IntT, StatusT, CancelT, LostT, MaxR)

```
// AppInstID - Application Instance ID is used for call-back routines
// Microtransactions - table of (StepID, HostID, SQL_Program)
// TokenPlan - Adjacency list (Status, CurrentID, NextID)
// TableSchemas - table of (ContractID, TableSchema, AccessControls)
// IntT - how long to wait for token during normal processing
// StatusT - how long to wait for status response
// CancelT - how long to wait for token after requesting cancel
// LostT - how long to wait for information about lost token
// MaxR - maximum number of times to reset the interaction timer
```

ApplicationID ← AppInstID
G ← (Microtransactions, TokenPlan)
InteractionTimer ← IntT
StatusTimer ← StatusT
CancelTimer ← CancelT
LostTimer ← LostT
MaxResets ← MaxR
InteractionID ← (HostID, ApplicationID, RandomDigitString)
Initialize containers and token catalog
for all (C, T, A) in TableSchemas **do**
  $K_C \leftarrow K_C \bigcup$ T
  $U \leftarrow U \bigcup$ (C, T, A, 0)
end for

```
// Initialize the token - see Definition 4.16
```

TokenID ← (InteractionID, RandomDigitString)
Ψ ←(TokenID, G, K, null, U, InteractionTimer, ready, null)
Write an "Initiate Interaction" record in the interaction log
Log the token plan in the interaction log
Deliver(Ψ, loopback)
Set timer ← InteractionTimer
Return InteractionID

Algorithm 6.4 Terminate_Interaction(Ψ)

Log Ψ in the interaction log
Write a "Terminate Interaction" record in the interaction log
Send "processing complete" message to application support layer

Similarly, if an interaction remains in the "Started" state for too long, the interaction timer will expire and Algorithm InteractionTimeout will be invoked. This algorithm issues a status request message and moves the interaction state to "Status sent".

Algorithm 6.5 CancelInteraction

```
// A cancel message is generated and broadcast to all of the
// peers listed in the outstanding token plan.
```

Issue cancel message to each peer in outstanding token plan
Set timer ← CancelTimer

Algorithm 6.6 InteractionTimeout

```
// A status request message is generated and forwarded to the
// first peer in the token plan of the currently outstanding token.
```

Issue status request to the first peer in the token plan of the outstanding token
Set timer ← StatusTimer

Algorithm 6.7 ReceiveStatus

```
// A status message has been received
```

if insufficient progress made OR resets in interaction log == MaxResets **then**
  CancelInteraction (Algorithm 6.5)
else
  Set timer ← InteractionTimer
end if

Upon receipt of a status response message, the interaction layer delivers the status information to the application support layer. At this point, end-user input or profile information may be used to determine whether to continue processing or abort processing based on the status information received from the interaction processing layer. Algorithm ReceiveStatus requests direction from the application support layer

Algorithm 6.8 Lost_Token

Broadcast a lost token message to all peers in the lost token plan
Set timer ← LostTimer
Wait for lost token responses until (timer expired or all token responses received)
Move to recovery with lost token responses

and based on the response, either extends the time to allow processing to continue and changes the state of the interaction back to "Started", or invokes algorithm CancelInteraction.

If the CancelTimer or StatusTimer expires, the Lost_Token algorithm is invoked. This algorithm broadcasts a lost token message to all participants in the interaction and accumulates as much information as possible about the status of the token before it was lost based on the lost token response messages received from participants. When sufficient information has been received to reconstruct the token or the LostTimer expires, the the recovery algorithm is invoked.

The Recovery algorithm is invoked if the token is returned with a status other than "Completed" or if the token is lost. Even for a lost token, part or all of the token may be reconstructable from the responses received by the Lost_Token algorithm by the time recovery is invoked. The recovery algorithm copies the token's log to the Interaction Log and forwards the log to the application support layer. The application support layer will then use the interaction plan to generate an alternative token plan and submit this new plan for execution at the interaction layer, or will opt for arbitration. If a new plan is forthcoming, the Recovery algorithm will construct a new token with this plan, deliver it to the first participant in the plan, and move the interaction state to the "Token sent" state. Otherwise, the state of the interaction will be moved to "Arbitration" and the interaction will terminate.

Algorithm 6.9 Recovery(Ψ)

Copy token log to the Interaction Log
Callback application layer instance ApplicationID with token log
Receive new token plan from application layer
Generate new token, $\Psi' \leftarrow \Psi$
Replace token plan on Ψ' with new token plan
Generate new TokenID for Ψ'
Log the new token plan in the interaction log
Deliver(Ψ', first destination in new token plan)
Set timer $\leftarrow$ InteractionTimer

6.3 Token Processing Layer

When a token is received by the token processing layer (whether it is received from the local interaction layer or from a peer token processing layer), the token is processed by Algorithm 6.10, Process_Token. This algorithm begins by ensuring that the token's ID is not on the lost token list. If the token is on the lost token list, it can be discarded. Otherwise, if the token has a status of "Failed", it can be handed to the local interaction layer for recovery. If the token is on the cancelled token list, the token status is set to "Failed" and the token is delivered to the originating peer.

If the token is not on the lost or cancelled list and the token status is not "Failed", then normal processing begins. First, the contents of containers on the token are decrypted using the decryption key in the appropriate contract. Next, containers on the token are moved into temporary tables in the DBMS of the peer. The SQL program associated with this microtransaction in the token plan is then submitted for execution on the peer's DBMS.

When DMBS processing completes, the temporary table contents are moved back onto the token's containers and encrypted. The return code for the DBMS is used to determine the next step to execute in the token plan. If there is no entry in the token plan for the return code, the token status is set to "Failed" and the token is returned to the originating peer for recovery processing.

If the last step in the token plan has successfully executed, the token status is set

to "Completed" and the token is delivered to the interaction layer.

If a decision is made to cancel an ongoing interaction, either due to time constraints or based on processing at a higher layer on the originating peer, the interaction layer issues a Cancel Request message to all sites participating in the interaction. Thus, the token processing layer is responsible for receiving and acting upon Cancel Request messages. The Process_Cancel_Request algorithm checks that the token to be cancelled is not pending in the site's outgoing tokens queue. If the token is queued for delivery, it is removed from the queue and sent back to the originating peer with a status of "Failed". Otherwise, the tokenID in the cancel token request message is added to the list of cancelled tokens.

Similarly, if token processing takes longer than expected, a status request message may be issued by the originating peer and the token processing layer is responsible for receiving and acting upon Status Request messages. The Process_Status_Request algorithm handles status request and status response messages at the token processing layer. The algorithm first checks the type of status message that has been received. If it is a status response message, the response is handed to the interaction layer for processing. Otherwise, the DMBS log of the peer is queried to determine the path followed by the token. If the token is being processed locally, a status response message is generated and sent to the originating peer. Otherwise, the status request message is forwarded along the same path followed by the token.

Finally, a token can be lost due to node failure. The interaction layer of the originating peer will detect that the token is lost based on the expiration of timers. The interaction layer will then issue token lost messages to each of the sites participating in the interaction. The token processing layer is responsible for handling lost token messages and responding to them with a lost token response message.

The Process_Lost_Request Algorithm handles token lost and token lost response messages at the token processing layer. If a token lost response is received, it is handed to the interaction layer for processing. If a token lost message is received, the ID of the token is added to the lost token list and a lost token response message

Algorithm 6.10 Process_Token(Ψ)

```
// Local data accessed include local contracts and the peer's DBMS
Starttime ← current local time
if TokenID on Lost list then
   Terminate processing
end if
if Token status is Failed then
   Queue token for delivery to local interaction layer
   Terminate processing
end if
if Contract expired then
   Set token status to Failed, token error code to Contract expired
   Queue token for delivery to originating peer
   Terminate processing
end if
if TokenID on Cancelled list then
   Set token status to Failed, token error code to Cancelled
   Queue token for delivery to originating peer
   Terminate processing
end if
Decrypt container using key from contract
Move containers on token into local database
Submit SQL program to DBMS front end for processing
Move containers from local database onto token
Encrypt container using key from contract
Reduce token counter by Starttime less current local time
Add token log entry
if Plan completed then
   Hand token to local interaction layer
   Terminate processing
end if
if No entry in token plan for code returned from SQL execution then
   Set token status to Failed
   Set token error code to code returned from SQL execution
   Deliver token to originating peer
   Terminate processing
end if
Deliver(Ψ, Next peer in token plan)
Terminate processing
```

Algorithm 6.11 Process_Cancel_Request

if Token queued for delivery to another peer **then**
  Remove token from queue
  Set token status to "Failed", token error code to "Cancelled"
  Queue token for delivery to originating peer
else
  Add the tokenID in the cancel token request message to the list of cancelled tokens in the appropriate contract
end if

Algorithm 6.12 Process_Status_Request

if MessageType == Response **then**
  Hand status response message to interaction layer
else if MessageID in Voided Message list **then**
  Discard Message
else
  Query local log to determine path the token followed
  if Token processing locally **then**
    Generate status response message
    Deliver Status Response to originating peer
  else
    Forward status request message along path taken by token
  end if
end if

Algorithm 6.13 Process_Lost_Request

if MessageType == Response **then**
  Hand lost token response to the interaction layer
else
  Add the tokenID in the lost token request message to the list of lost tokens in the appropriate contract
  Generate a LostTokenResponse with the contents of the temporary tables and a copy of the local logs that include the lost tokenID
  Deliver the LostTokenReponse to the initiating peer
end if

is generated and sent to the originating peer. This response message includes any information the site has concerning the last known state of the token, including the "stale" values stored in temporary tables used to transfer data to and from containers on the token.

Chapter 7

Conclusions and future work

In this thesis, we have proposed a new model for the coordination of interactions in a peer-to-peer process-oriented programming environment. The model is motivated by the need for an interaction coordination model that allows end-user systems to act as peers in a peer-to-peer environment in order to support extensive individualization of interactions on a variety of devices while preserving end-user privacy.

A new model of transaction coordination is needed to allow end-user control of end-user information while still allowing participation in distributed transactions. The token-based peer-to-peer interaction coordination (TPIC) model described in this thesis fulfills this requirement. This new model is fundamentally different from other distributed transaction coordination models in that it does not rely on global state to effect the coordination of distributed activities.

The TPIC model makes use of a mobile, disposable token for coordination and transfer of data among peers. Pairs of peers must enter into contracts before they can participate in TPIC model interactions. Once contracts have been established, an interaction plan can be generated. An interaction plan is a directed acyclic graph of microtransactions to be executed in sequence at the peers participating in the interaction.

The approach taken in the TPIC model is to serialize the actions to be performed at different sites based on a plan that resides on a token, and to ship data among

sites on this token. This allows processing at a single site to execute independent of the state of any other sites, since all of the data the site needs to access are made local to the site by the presence of the token.

The processing that occurs at a single site is termed a microtransaction (similar to a subtransaction in a multidatabase environment), and the execution of a microtransaction moves the site's DBMS from one consistent state to another consistent state. Only one microtransaction on a token can be active in the system at any given time, since the execution of microtransactions is serialized by the token plan.

Important assumptions underlying the model include:

1. The model does not rely on access to global state
2. The model does not support data replication
3. Each peer maintains a DBMS
4. A peer DBMS is a pre-existing database and its software cannot be modified in support of the model
5. A peer DBMS views requests initiated by a remote peer under the TPIC model no differently than requests that are generated locally
6. There is a reliable message-passing infrastructure in place

The three most important properties of the TPIC model are:

1. The TPIC model is non-blocking
2. The TPIC model does not rely on global state
3. The TPIC model provides a mechanism for enhancing end-user privacy

The TPIC model is a non-blocking model, in that locks are only held by a microtransaction while it is executing at a given peer. Each microtransaction forms a complete transaction on the peer where it executes, moving the peer's DBMS forward from one consistent state to another consistent state. Data replication is not

supported by the TPIC model. Instead, data is moved onto and off of the token. Since only a single copy of each data item is supported, data is shared among peers by moving the data to the peer that needs it. The token assumes the role of the transaction coordinator, and containers within the token allow data to be moved among peers. Since each peer's DBMS is left in a consistent state when the token leaves a peer, recovery from failures can be quite flexible as none of the participating peers are blocking pending recovery.

The TPIC model does not rely on global state for the coordination of distributed interactions, hence there are no interdatabase constraints enforced by the model. Instead, contract terms that ensure consistency across peers are negotiated and enforced. Each interaction is governed by its own coordinator and the microtransactions to be performed are serialized on a token. Each microtransaction forms a complete transaction on a peer's DBMS, thus ensuring that the peer's DBMS will be left in a consistent state regardless of whether the microtransaction execution commits or aborts.

The TPIC model provides a mechanism for enhancing end-user privacy. Placing end-user information under end-user control presents problems for existing distributed transaction processing systems, since access to this data is required for processing to occur. Using the token as a vehicle for moving data from peer to peer allows the end-user to maintain their own information and to control how it will be shared with other peers. Contract terms negotiated before interactions occur allow the end-user to enforce their own privacy policies.

If an interaction completes successfully, it is guaranteed to move the state of each peer involved in the interaction from one contractually conformant state to another contractually conformant state, meaning that all of the contract terms of each of the contracts governing the interaction will be fulfilled. In the event that the interaction cannot successfully complete, the logs of the interaction will be sent to an arbitration process for resolution. Plan generation and plan selection tools should be used to reduce the likelihood of arbitration by providing flexible, alternative plans

when potential failures are anticipated.

Further work is required to investigate performance and scalability of the TPIC model. It is difficult to draw comparisons directly between existing distributed database implementations and the TPIC model. In existing databases, clients issue transactions to be processed on a limited number of servers. In contrast, the TPIC model replaces clients and servers with peers, and data and processing are distributed among all of the peers. This results in a model with potentially hundreds of millions of peers, each with a very limited amount of data.

Future work of an interdisciplinary nature is required for development of tools in support of the contract negotiation and establishment and plan generation. Tools for domain experts are needed that allow contract terms to be defined in a standardized way and that allow plans to be specified.

While the process-oriented programming environment inspired the placement of data and control on the token, the TPIC model does not depend on the process-oriented programming environment. The TPIC model could be used with existing database environments without significant modification to the model. Future work is needed to investigate how the TPIC model can be integrated into existing middleware and workflow environments to create hybrid systems that link the existing infrastructure to the peer-to-peer paradigm.

Future work will investigate implementation issues with respect to the use of the token counter to prioritize token processing at peers. The token counter could also be used to assign priorities during transaction processing in a peer's DBMS, allowing tokens with ample time remaining to be selected preferentially during deadlock resolution. Extensions to the model could also involve the use of the token counter at the token processing layer to automatically cancel tokens whose counter has expired.

Each peer in the TPIC model is treated as an autonomous entity that provides access to DBMS facilities. Implementation issues may arise when peers involved in a TPIC interaction use middleware facilities such as CORBA or DCOM to access distributed resources and/or legacy database systems. While the TPIC model has

been designed to accomodate peers with any DBMS (including distributed or legacy systems), implementation issues may force extensions to the model in order to accomodate the types of failure these database systems can encounter.

The TPIC model as presented in this thesis relies on bilateral contracts between pairs of peers. Extending the model to use multi-lateral contracts should be investigated, as it would allow containers to be shared by more than two participants.

A fully functional prototype of the TPIC model will be implemented as part of the KALI project.

Acknowledgements

Many thanks to my supervisor, Dr. Jacob Slonim, committee members, Dr. Carl Hartzman, Dr. Michael McAllister, and Dr. Nick Cercone, and external examiner, Dr. John Mylopoulos. This research was funded in part by grants from the National Sciences and Engineering Research Council of Canada, the Killam Foundation, and the IBM Centre for Advanced Studies.

Bibliography

[1] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor, and C. Mohan. Advanced transaction models in workflow contexts. *Proceedings of the Twelfth International Conference on Data Engineering*, pages 574–581, 1996.

[2] Stan Zdonik et al Avi Silberschatz. Strategic directions in database systems - breaking out of the box. *ACM Computing Surveys*, 28(4):764–778, 1996.

[3] M.A. Bauer, N. Coburn, D.L. Erickson, P.J. Finnigan, J.W. Hong, P.-A Larson, J. Pachl, J. Slonim, D.J. Taylor, and T.J Teorey. A distributed system architecture for a distributed application environment. *IBM Systems Journal*, 33(3):399–425, 1994.

[4] D. Bell and J. Grimson. *Distributed Database Systems.* Addison-Wesley, 1992.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *Conncurrency Control and Recovery in Database Systems.* Addison-Wesley Longman, 1987.

[6] Philip A. Bernstein and Nathan Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 283–300, Montreal, Canada, October 1980.

[7] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[8] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proc of ACM SIGMOD Workshop on Data Description, Access, and Control*, pages 249–264, Ann Arbor, Michigan, USA, May 1974.

[9] T. Chiasson, K. Hawkey, M. McAllister, and J. Slonim. An architecture in support of universal access to electronic commerce. *Journal of Information and Software Technology.*, 44:279–290, 2002.

[10] Workflow Management Coalition. Workflow reference model. Technical report, 1995.

[11] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[12] Microsoft Corporation. http://www.microsoft.com/com/tech/DCOM.asp [Last Visited: March, 2003].

[13] G. Coulson, G.S. Blair, N. Davies, P. Robin, and T. Fitzpatrick. Supporting mobile multimedia applications through adaptive middleware. *IEEE Journal on Selected Areas in Communications*, 17(9):1651–1659, 1999.

[14] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1983.

[15] P. Dickenson and J. Ellison. Getting connected or staying unplugged: The growing use of computer communications services. *Services Indicators*, pages 17–34, 1999.

[16] H. Duran and G.S. Blair. A resource management framework for adaptive middleware. *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 206–209, 2000.

[17] F. Eliassen, A. Andersen, G.S. Blair, F. Costa, G. Coulson, V. Geobel, O. Hansen, T. Kristensen, T. Plagemann, H. Rafaelsen, K. Saikoski, and W. Yu.

Next generation middleware: Requirements, architecture, and prototypes. *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 60–65, 1999.

[18] W. Emmerich. Software engineering and middleware: A roadmap. *Proceedings of the 22nd International Conference on The future of Software engineering*, pages 117–129, June 2000.

[19] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, 1976.

[20] H. Garcia-Molina and K. Salem. Sagas. *Proceedings of ACM SIGMOD*, pages 249–259, 1987.

[21] Gnutella. http://www.gnutella.com [Last visited: March 2003].

[22] P. Grefen, B. Pernici, and G. Sanchez, editors. *Database Support for Workflow Management: The WIDE Project.* Kluwer Academic Publishers, 1999.

[23] Object Management Group. Common object resource broker architecture. http://www.corba.org/ [Last Visited: March, 2003].

[24] J. Hamilton. Operational data storage unification. Technical Report CS-97-16, University of Waterloo, July 1997.

[25] KaZaa. Sharman networks ltd. http://www.kazaa.com [Last visited: March 2003].

[26] H. T. Kung and John J. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, June 1981.

[27] SUN Microsystems. Enterprise javabeans.

[28] Thomas Mikalsen, Stefan Tai, and Isabelle Rouvellou. Transactional attitudes: Reliable composition of autonomous web services. *WDMS 2002*, June 2002.

[29] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[30] E. Moss. *Nested Transactions, An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[31] J. Moss. Nested transactions and reliable distributed computing. *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, 1982.

[32] OASIS. Business transaction protocol: An oasis committee specification. version 1.0. http://www.oasis-open.org/committees/business-transactions/ [Last visited: March 2003].

[33] OASIS. Organization for the advancement of structured information standards. http://www.oasis-open.org [Last visited: March 2003].

[34] M. Papazoglou. Web services and business transactions. *WWW Journal*, pages 49–91, March 2003.

[35] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, 1995.

[36] F. Siqueira and V. Cahill. An open qos architecture for corba applications. *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 328–335, 2000.

[37] D. Skeen. A quorum-based commit protocol. In *Proceedings of the sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, Pacific Groove, CA, USA, February 1982.

[38] R.G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[39] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.

[40] R. Strom. A comparison of the object-oriented and process paradigms. *SIGPLAN Notices*, 21(10):88–97, 1986.

[41] R. Strom, D. Bacon, A. Goldberg, A. Lowry, D Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing.* Prentice Hall, 1991.

[42] N. Venkatasubramanian. Compose—q - a qos-enabled customizable middleware framework for distributed computing. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware*, pages 134–139, 1999.

[43] W3C. Simple object access protocol. 2000. Availble, http://www.w3.org/TR/SOAP/ [Last Visited: March 2003].

[44] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.