The Open University of Israel
Department of Mathematics and Computer Science

TOPS

Transaction Oriented Publish-Subscribe and its Contribution to Implementing Replication

Thesis (Final Paper) submitted as partial fulfillment
of the requirements towards an M.Sc. degree in Computer Science
The Open University of Israel
Computer Science Division

By
Yosef Shatsky

Prepared under the supervision of Prof. Ehud Gudes

March 2010

Acknowledgements

My thanks to my advisor, Professor Ehud Gudes, for proving me with technical guidance and support vital to the success of my thesis. Additionally, I would like to thank Luis Vargas of Cambridge University for supporting my work by providing source code and information regarding HTS.

Contents

Abstract **1**

1 Introduction **3**

1.1 General 3

1.1.1 Middleware 3

1.1.2 Event-Based Systems 4

1.1.3 Distributed Transactions 5

1.2 Main Contributions 5

1.3 Thesis Structure 6

2 Background and Related Work **8**

2.1 Introduction to Publish/Subscribe 8

2.2 Hermes 11

2.3 Transactional Publish/Subscribe and HTS 11

2.3.1 HTS Concepts 12

2.3.1.1 Census Phase 12

2.3.1.2 Transaction Phase 13

2.3.1.3 Commitment Phase 13

2.3.2 HTS Architecture 13

2.3.3 Transaction Context 14

2.3.4 Additional HTS Features 15

2.4 Introduction to Database Replication 16

2.5 Related Work 17

2.5.1 Pub/sub Transactions 17

2.5.2 Database Replication using Pub/Sub 18

3 Analysis of HTS – The Application Simulator **19**

3.1 Requirements & Requirement Analysis 19

3.1.1 Requirements Analysis: Use-Case Diagram 20

3.1.2 Use-Case 1: Successful Transaction 22

3.1.3 Use-Case 2: Census Not Fulfilled 23

3.1.4 Use-Case 3: Commit Fails 24

3.1.5 Use-Case 4: Multiple Successful Transactions 25

3.1.6 Interaction with HTS and API Requirements 26

3.2 Simulator Design 30

3.2.1 Software layers 30

3.2.2 Class Diagrams 32

3.2.3 Threads 35
4 TOPS – Transaction Oriented Publish/Subscribe 36
4.1 Motivation for TOPS 36
4.1.1 Transactions with Multiple Publishers 37
4.1.2 Transaction Initiator is not Required to Participate 37
4.1.3 No Need to Explicitly Subscribe to Transactions 38
4.1.4 Automatic Delivery of Any Event-Type 39
4.1.5 Test of Constraints at Commit 40
4.1.6 Support for Local Transactions 41
4.1.7 Motivation Conclusion 41
4.2 Transactions Supported 42
4.2.1 Local Transactions 42
4.2.2 Distributed Transactions 43
4.2.3 Mixing Distributed and Local Transactions 43
4.3 Event Dissemination 44
4.4 Transaction Properties 45
4.4.1 Participation of Initiator 45
4.4.2 Access 45
4.4.3 Scope 45
4.4.3.1 Scope Limitation 46
4.4.4 Constraints 47
4.5 TOPS Architecture 49
4.5.1 Transaction Management 49
4.5.2 Component Configurations 51
4.5.3 Event Sequencing 55
4.5.4 Transaction Context 55
4.6 Transaction Processing 56
4.6.1 Transaction Establishment 56
4.6.2 Transaction Execution 61
4.6.3 Transaction Commitment 63
4.7 Application Programming Interface 63
4.7.1 The Influence of TOPS Concepts on the API 64
4.7.2 Use Case #1 64
4.7.3 Use Case #2 66
5 Applying TOPS to Replication 70
5.1 Methods of Replication 70

5.1.1 Asynchronous Methods of Replication....71
5.1.2 Synchronous Methods of Replication....72
5.1.3 Replication Methods Conclusion....73
5.2 Implementing Replication Using TOPS....73
5.2.1 Asynchronous Peer-to-Peer....74
5.2.2 Asynchronous Primary Site....75
5.2.3 Synchronous Read-Any Write-All....75
5.2.4 Synchronous Quorums/Voting....76
5.2.5 Combined Synchronous and Asynchronous....77
5.2.6 Conclusion....78
5.3 Advanced Replication methods....78
5.3.1 The BackEdge Protocol....78
5.3.2 Replication with Serializability....82
5.4 Summary....86
6 Conclusion....87
6.1 Future Work....87
7 References....89
8 Appendix A – HTS Application Simulator....92
8.1 Implementation of Simulator....92
8.1.1 Development Environment....92
8.2 User Interface Design....92
8.2.1 Main Window....92
8.2.2 Client Window- General....93
8.2.3 Client Window- Advertise and Register Tab....94
8.2.4 Client Window- Transaction Status Tab....96

List of Figures

Figure 2-A: Pub/Sub Scheme....10
Figure 2-B: HTS/Hermes Middleware....12
Figure 2-C: HTS Transaction Management....14
Figure 2-D: HTS Transaction Context....15
Figure 3-A: Use-Case Diagram (Manager)....21
Figure 3-B: Use-Case Diagram (Publisher and Subscriber)....21
Figure 3-C: Sequence Diagram- Simulator and HTS Interaction....29
Figure 3-D: Software Layers....30
Figure 3-E: Application Simulator Class Diagram....34
Figure 4-A: Transactions with Multiple Publishers....37
Figure 4-B: Transactions with Initiator not Participating....38
Figure 4-C: No Need to Explicitly Subscribe to Transactions....39
Figure 4-D: Automatic Delivery of Any Event-Type....40
Figure 4-E: Test of Constraints at Commit....40
Figure 4-F: Support for Local Transactions....41
Figure 4-G: Local transactions vs. distributed transactions....42
Figure 4-H: Optimization of Rendezvous Node Multicast....44
Figure 4-I: Legal Scopes According to Transaction Type and Scope Limitation....46
Figure 4-J: TOPS Architecture....49
Figure 4-K: Communication in Tx Manager Configurations....52
Figure 4-L: Tx Manager Configurations....53
Figure 4-M: Component Configuration Scenario....54
Figure 4-N: TOPS Transaction Context....56
Figure 4-O: API Use Case #1 Sequence Diagram....66
Figure 4-P: API Use Case #2 Sequence Diagram....69
Figure 5-A: Primary Site Replication Isolation Violation....72
Figure 5-B: Example Replication Copy Graph....79
Figure 5-C: DAG(WT) Copy Tree....80
Figure 5-D: Example Replication Copy Graph with Back-Edge....80

List of Tables

Table 4-A: Transaction Scope 47
Table 4-B: TOPS API.....64
Table 8-A: Development Environment Specification 92

Abstract

Publish/subscribe is a common messaging paradigm used for asynchronous communication between applications, in which a publisher is a *message provider* and a subscriber is a *message consumer*. Publish/subscribe is scalable and is tolerable of frequently changing subscriber lists, mainly due to decoupling of the publishers from the subscribers. When one publishes a message, he is unaware of who will receive the message, where they are located and when they will receive the message. All these issues are managed by the middleware. The publish/subscribe paradigm is applicable to a wide range of applications. Recent studies have shown that publish/subscribe is suitable even for applications with extreme conditions, such as real-time applications requiring nothing less than top performance and cellular applications with most difficult connectivity and bandwidth conditions.

In *synchronous publish/subscribe* a distributed transaction is used to have all subscribers participating in the transaction receive and act upon the published messages in an atomic manner. Synchronous publish/subscribe middleware exist but are less common than their asynchronous counterparts for two main reasons. The first is that message dissemination involves larger delays. These delays exist because all participants must wait to commit a message till all other subscribers in the transaction are prepared to commit. In effect, the slowest subscriber is the one pacing the transaction. The second reason is that resources remain locked for a much longer period of time. All subscribers in the transaction will lock resources before issuing an update of data. Since all subscribers commit together, resources remain locked while waiting for remote subscribers to complete their work. During this period of time, all transactions attempting to access the locked resources will be forced to wait.

Hermes Transaction Service (HTS) is such a middleware, which is capable of treating a group of publications as a transaction. In this paper, we propose a design for a new transactional publish/subscribe middleware, based on HTS. Accordingly, we name the middleware TOPS – Transaction Oriented Publish/Subscribe. In order to justify using HTS as a starting point for TOPS, an application simulator was built over HTS. This simulator enables using HTS from an application's point of view. By using this simulator we were capable of studying HTS in depth, learning its exact behavior and thereby extracting valuable conclusions which were used towards designing TOPS. These conclusions provided a significant contribution to TOPS's design and capabilities.

To further demonstrate the advantages of the TOPS middleware, we present how different strategies of replication may be implemented all using the TOPS middleware

proposed. Replication is the process of managing more than one copy of the same data on different servers in order to gain redundancy and performance. As in publish/subscribe, replication may be synchronous (i.e. all replicas update atomically) offering high data integrity and low performance, or asynchronous (i.e. one replica updates immediately and all other replicas update eventually) offering low data integrity and high performance. Implementation of asynchronous replication using publish/subscribe is straightforward and currently exists in several database solutions. However, using our design of a transactional middleware, the implementation of synchronous replication becomes quite simple as well. We detail how several basic methods of replication can be implemented using the TOPS middleware.

Beyond the scope of TOPS we give a theoretical discussion about how more advanced methods of replication may be implemented with the help of a transactional publish/subscribe middleware. Advanced replication algorithms attempt to maximize both performance and data integrity usually by restricting the sequence of update propagation or by limiting the use of distributed transactions to a small set of situations. We argue that the applicability of implementing algorithms at this level using publish/subscribe, is algorithm specific. To back this claim, two sample replication algorithms are presented, one applicable to publish subscribe and one not.

1 Introduction

1.1 General

Computer systems have come a long way since the day of using a point-to-point cable for facilitating communication between multiple entities. Once computer networks came in to play, the physical connection became much more flexible, however, data sharing at the application level largely remained point-to-point and was implemented directly over TCP. Even though multicast was available, management of the multicast groups was not automatic. As networks became commonplace, large scale applications became highly distributed and required not only a generic network structure but also a generic means of sharing application data, at a higher level than what the operating system provides. Consider an application coordinating the activity of the New York Police Department. At one end, the application is run at the police head quarters which contains powerful servers, and at the other end are patrolling officers or vehicles carrying computing platforms with limited resources. Information is exchanged between many entities, officers, citizens calling the emergency lines, other emergency services such as the fire department and fellow police departments. Information is required to flow in a timely manner only to entities to which the data is of interest. Such an information system requires much more than the standard set of TCP/IP protocols.

1.1.1 Middleware

As mentioned above, large-scale distributed enterprise applications must deal with a variety of challenges regarding connectivity of their components. Such applications often span several operating systems and have different means of communication, each with its own throughput, latency and offline time. In the past few years it has become common for applications to support cellular phone clients. All sites must adapt to changes in availability of sites, for instance when a backup server has kicked in to replace a primary server which is down. Additionally, different components may require different communication schemes (i.e. push, pull or notification, discussed in the following section). To free the application of the burden of dealing with all these issues, a software layer named *middleware* is placed “in the middle” between the application and the operating system. The middleware provides connectivity and services to applications in a heterogeneous environments. Using a middleware, applications developers can focus more on the business logic and less on technical implementation details.

Different types of middleware vary in the way they share data:

- Message Oriented Middleware (MOM) shares data by means of sending messages. Messages are sent to the recipient and are stored there until he is available to read them.
- Remote Procedure Calls (RPC) allows the invocation of a procedure on a remote site. Parameters passed to the procedure are transferred to the remote site.
- Objects sharing, suitable for object oriented environments.
- Transaction processing, allowing applications to make use of distributed transactions.

Since a more generic middleware will support more types of application, the ambition is to include as many features as possible in the middleware- synchronous (blocking) and asynchronous (non-blocking) remote invocation, messaging, and transactions.

1.1.2 Event-Based Systems

In an event-based system, there are *sensors* that produce events, and event consumers that perform an action when events occur. An event is any occurrence of interest. This occurrence produces an event that includes information about the specific occurrence. The event is sent asynchronously into the event-based system which is responsible for having the event reach all relevant consumers. The producer and consumer are unaware of one another; the connection between them is managed entirely by the event-based system. Middleware implementing an event-based system associate event producers and consumers using a **publish/subscribe** paradigm. Consumers, now called subscribers, tell the middleware what kind of information they are interested in (not who produces it). Producers, now called publishers, tell the middleware what kind of information the event being published contains. The middleware assumes the difficult task of sending the event only to the relevant subscribers based on the data of interest they specified during subscription.

The publish/subscribe paradigm came to replace the request/reply paradigm for certain system architectures. In the request/reply paradigm there is a server capable of providing services and any client interested in these services must be aware of this server and contact it. For example, the request/reply model is not suitable when many servers exist or when the server may change over time. In publish/subscribe the event is delivered till the consumer, without the consumer needing to contact, or even be aware of, the producer.

Interestingly, a good example of the usefulness of notification can be brought from the database world. The traditional use of a database is by a client requesting data from the

database using queries, as in the request/reply paradigm. However, for quite a while already, *active databases* exist that are capable of notifying at the occurrence of an event, and taking action, instead of waiting for the application to run another query.

CORBA (Common Object Request Broker Architecture) and JMS (Java Messaging Service) are two standards addressing notification systems. Both support point-to-point and publish/subscribe communication. A few examples of implemented publish/subscribe systems are: TIBCO rendezvous (commercial), IBM's WebSphere Business Integration Event Broker (commercial), Hermes [19] and Siena [3].

1.1.3 Distributed Transactions

Transactions are a fundamental building block of applications. As applications became distributed, the underlying transaction subsystem has become distributed as well. Distributed transactions are required to lock resources and apply changes to more than one site. Each site executes a sub-transaction and a centralized transaction manager coordinates between all sub-transactions. Transaction management requires communication between sites. For instance, the transaction manager must instruct each sub-transaction what it should do, for instance when it should commit. Since transactions are an elementary feature and since communication is necessary, it makes sense to have a middleware provide such transactional services.

The X/Open Company has published a model for support of distributed transactions that has been adopted by many transactional middleware. Their publications include an architecture and specification that cover the necessary interfaces and behavior required from the implementation.

The .NET and Java development frameworks both provide support for application development with distributed transactions.

1.2 Main Contributions

Our first main contribution is the detailed analysis of HTS [26]. HTS is a recently developed (2005) advanced middleware supporting content-based transactional publish/subscribe with a respectable set of features such as public and private transaction, compensatable clients and more. HTS is build on top of Hermes [19], a publish/subscribe middleware, and augments it with transactional capabilities. We analyzed HTS at the theoretical level and at the implementation level. To aid this analysis, an application simulator was developed solely for this purpose. This detailed work yielded several interesting and most significant results which are detailed in

section 4.1. The simulator and the conclusions reached with its assistance are the first contribution of this work.

The second main contribution is the design of an improved transactional pub/sub middleware (TOPS). TOPS benefits from more natural integration of transactions into the pub/sub paradigm. This integration allows TOPS to overcome several weaknesses of HTS. For example, subscribers are no longer required to know in advance what event types a transaction will contain. TOPS' design also provides several new features such as local transactions (see section 4.2.1). In our opinion, the work in this paper brings the research on the subject of transactional pub/sub a step forward.

The third main contribution is the integration of TOPS and data replication. Replication solutions commonly use pub/sub to implement asynchronous replication. Using TOPS, synchronous replication can be implemented as well, all using the same middleware. This paper shows how, by using TOPS, several methods of synchronous and asynchronous replication can all be implemented over a single transactional pub/sub middleware. A paper describing TOPS was published recently in DEBS '08, the leading conference on the subject of event based systems.

1.3 Thesis Structure

The structure of the rest of this thesis is as follows:

Chapter 2- Background and Related Work

This chapter contains background information and related work on the subjects of publish/subscribe, the transactional aspect of publish/subscribe, and database replication. The transactional aspect of publish/subscribe is conferred by reviewing concepts, architecture and features of HTS, an existing transactional pub/sub middleware. In addition to background information, this chapter emphasizes specific points that are relevant to the design of TOPS, either because TOPSs utilizes them or because TOPSs improves upon them.

Chapter 3- Analysis of HTS – The Application Simulator

As part of our research on the subject of pub/sub, in order to evaluate the strengths and weaknesses of the design of HTS, an application simulator was developed over HTS. This chapter presents the requirements analysis and design of this simulator. A more detailed description of the simulator is given in Appendix A. The findings of this analysis fuel the motivation for the design of TOPS as given in the next chapter. It is not necessary to read this chapter in order to understand the rest of the thesis.

Chapter 4- TOPS – Transaction Oriented Publish/Subscribe

In this chapter TOPS, the design of a new pub/sub middleware architecture, is presented. At first, the motivation for a new architecture is discussed. The chapter continues with defining the behavior of transactions, components and architecture of TOPS. For conclusion, an API is defined and demonstrated.

Chapter 5- Applying TOPS to Replication

The capabilities of TOPS are demonstrated in this chapter by showing how TOPS can be used as a platform for database replication. This chapter begins by presenting common methods of replication. Next, it is shown how the methods of replication can be implemented using the TOPS middleware.

Chapter 6- Conclusion

This chapter contains a few concluding remarks regarding this work.

Chapter 7- References

List of papers/sources quoted throughout this document.

Chapter 8- Appendix A – HTS Application Simulator

This appendix describes the implementation of application simulator including GUI screenshots.

2 Background and Related Work

In this chapter we discuss background information and related work in three areas:

- Publish/Subscribe
- Transactional publish/subscribe.
- Replication

We give an especially detailed description of Hermes and HTS.

2.1 Introduction to Publish/Subscribe

In complex applications, a message oriented middleware (MOM) is commonly used to satisfy the communications requirements of the application. A publish/subscribe (pub/sub) [6] middleware is a type of MOM suitable for environments with changing message traffic and connectivity. Pub/sub does not directly connect the message sender and receiver; rather it connects them to the middleware which dynamically decides which subscribers will receive each message.

Publish/subscribe is a message (event) driven scheme for distributing events between event providers (publishers) and event consumers (subscribers). Publishers provide events of a certain type or with certain content to the middleware, and subscribers independently request to receive events of certain type or content. The pub/sub system is responsible for notifying subscribers of published events matching the criteria they specified in their subscriptions.

The key strength of pub/sub is the decoupling of publishers and subscribers.

- Space decoupling – A publisher can publish an event without needing to worry about who must receive the event and where they are located within the network. Additionally, when many publishers publish events of similar type and content, subscribers will receive the events without knowing who the providers are and without being aware of the fact that multiple providers exist.
- Time decoupling – The subscribers are not limited to receiving events at the time they are published, or at the same time as other subscribers. Subscribers may be offline when the publisher publishes the event, and this fact is of no concern to the publisher. The pub/sub middleware is responsible for eventual delivery of events to the appropriate subscribers.
- Synchronization decoupling – Production of an event and its consumption are asynchronous operations. The significance of this is that the publisher sends an event, the request is accepted and control is returned to the calling function. The

event is sent asynchronously and therefore the publisher is not blocked while the event is being sent. Asynchrony applies to the subscriber in a similar manner.

Due to the reasons above, pub/sub scales easily and adapts transparently to changes in subscriptions and network topology.

A pub/sub system may be topic-based, in which case events posted to a topic will be received by all subscribers to the topic, or content-based in which case subscribers will receive any event containing information of interest according to the subscriber's subscription.

Unlike point-to-point or multicast communication, in which the sender must determine who the recipients are, in pub/sub the recipients themselves enlist (conditional to proper security privileges) to specific types of content/event types, and will be notified of new messages directly by the middleware without the publisher being involved. This is obtained by having event brokers disseminate events based on the existing subscriptions. Pub/sub also provides anonymity between publishers and subscribers.

Publishers and subscribers are not necessarily two separate entities. A single entity may publish one type of event and subscribe to another, or even subscribe to an event type it itself may publish, which is likely to be when more than one publisher exists for the same event type.

Conceptually, a pub/sub middleware exposes two interfaces, one for the publisher and one for the subscriber (Figure 2-A). The publisher's interface consists of the operations:

- advertise() – which notifies the middleware of the publisher's intension in publishing events of a certain type.
- publish() – publish an event 'event' of type 'event type' to all enlisted subscribers.

The subscriber's interface consists of:

- subscribe() – for subscribing to event type 'event type'. The middleware will notify the subscriber when an event of this type arrives using a callback function/object.
- unsubscribe() – to stop being notified of events of type 'event type'.

When using topic-based pub/sub the event type is simply the topic. When using content-based pub/sub the message type may enclose complex rules for filtering the content.

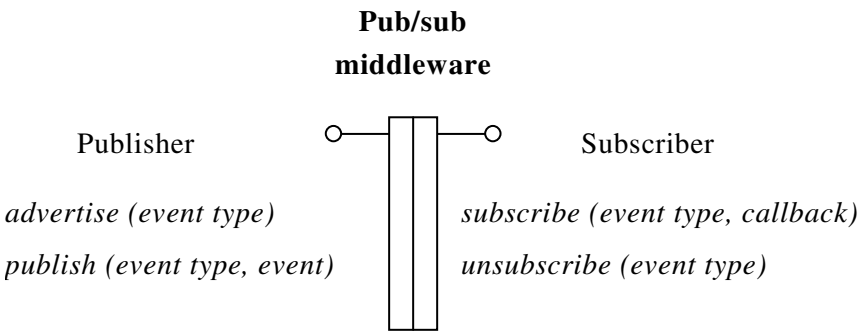


Figure 2-A: Pub/Sub Scheme

In [11], a survey of event notification services is given. From this survey, a few systems supporting pub/sub are mentioned here. TIBCO rendezvous, by TIBCO software, is a commercial product providing topic-based pub/sub. It is capable of running on several platforms (including Windows and different flavors of Unix/Linux) and can be used from within many programming languages (Java, C++ and others). The NASDAQ trading floor software is implemented using TIBCO rendezvous. IBM WebSphere MQ is a messaging platform by IBM. It too supports many platforms. WebSphere MQ supports only topic-based pub/sub, but it can be complemented with an add-on component that supports content-based pub/sub. Siena [3] and Hermes [19] are two research prototypes aimed at Internet scale applications. Both support content-based pub/sub. Hermes provides event-type checking and event-type hierarchies. Hermes is further detailed in Section 2.2 as it is of relevance to our work.

Pub/sub systems may be extended to support distributed transactions [13]. In the context of pub/sub, a transaction is a set of operations defined by a publisher which will either be applied by all subscribers or not applied by any subscribers. Distributed transactions are a powerful tool for applications, but this comes at the cost of locking resources at the participants of the transaction for longer periods of time and it also introduces latency due to the two-phase commit required by the transaction [12]. Security and permissions become a critical issue when working with transactions since a single misbehaving subscriber may continuously cause transactions to abort.

2.2 Hermes

Hermes [19] is a content-based pub/sub middleware consisting of *event clients* and *event brokers*. Event clients are the publishers and subscribers, event brokers represent the middleware itself and are connected to each other in an unrestricted topology. The brokers provide the clients with the ability to define an event-type, advertise an event-type, publish an event, and subscribe to an event-type. The brokers are responsible for the distribution of publications to all the appropriate subscribers.

Hermes is actually a hybrid of topic-based and content-based pub/sub. Topics are used as in standard topic-based pub/sub. In addition to a topic, publications may contain attributes describing the content of the event, and subscriptions may contain attributes describing the content of interest. The attributes are used to filter out events only within the topic.

Routing of events is accomplished by defining a *rendezvous node* for each event-type. The rendezvous node for each event type is an event broker whose id is closest to the hash value of the event-type's id. All advertisements, subscriptions and publications for a specific event-type will reach the same rendezvous node, thereby providing a central connection point between publishers and subscribers.

2.3 Transactional Publish/Subscribe and HTS

A pub/sub middleware may be augmented with transactional capabilities. Transactions are commonly a necessity for distributed applications, particularly for large scale applications. Messaging and distributed transactions were, at first, separate capabilities. As applications became more demanding, it became necessary to provide message delivery and message handling together within a transaction [16], [24], [9]. Transactional content-based pub/sub was introduced in [13]. Note that pub/sub distributed transactions do not come to replace classic distributed transactions. Pub/sub distributed transactions are very suitable when all participants require the same data and/or perform the same work, since communication is very efficient even for a large number of participants. On the other hand, if the distributed transaction consists of only few participants, each requiring different data, pub/sub will not be beneficial.

We based our work on the architecture of an existing transactional pub/sub middleware, which allows us to focus on issues relevant to transactions and avoid less-related issues such as security, transaction logs, error handling, event matching, etc. The middleware chosen was Hermes Transaction Service HTS (HTS) [26]. HTS was chosen for the following reasons: 1) It supports content based pub/sub, 2) It supports compensatable and non-compensatable clients, 3) It contains a conditional census phase, 4) It provides type-based pub/sub, and 5) It supports public and private transactions. Issues not

addressed in TOPS are as defined in HTS. This section describes the features and architecture of HTS as a prerequisite to fully understanding TOPS.

2.3.1 HTS Concepts

HTS is a transaction service built upon Hermes (Figure 2-B). Unlike database transactions which fulfill all four ACID (Atomicity, Consistency, Isolation, and Durability) properties, HTS supports *pub/sub transactions* that provide atomic execution over all clients participating in the transaction. Within a transaction, if any client fails to lock resources or encounters some other error, the middleware will initiate a rollback at all the other clients participating in the transaction.

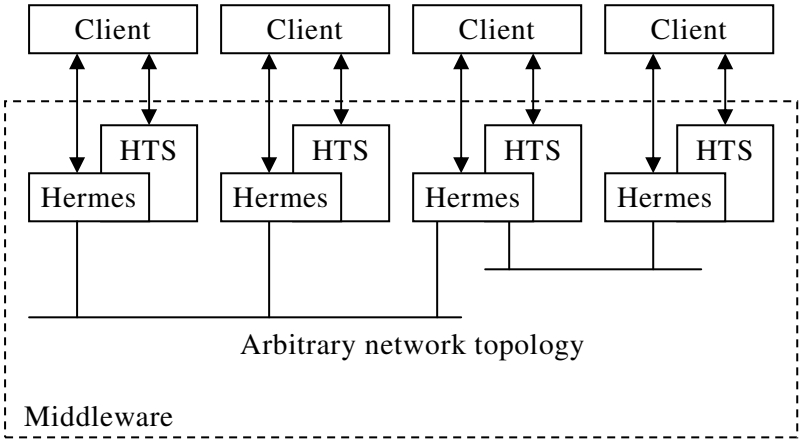


Figure 2-B: HTS/Hermes Middleware

In HTS, pub/sub transactions consist of the following three phases:

2.3.1.1 Census Phase

The census phase determines which subscribers will participate in the transaction. Standard pub/sub provides time decoupling between publishers and subscribers; therefore, events are sent to all subscribers- including ones that may be offline. Since a transaction requires subscribers to be currently active, an event is sent to all subscribers asking who is interested in participating in the transaction. Interested subscribers enlist to the transaction by replying to the census event and including their id hashed using a one-way hash function. At the publisher side, HTS maintains a list of hashed subscriber ids who answered positively. This list will be published in the next phase and each subscriber will look for his own hashed id to determine if it was enlisted in the transaction. When initiating the census phase the publisher may define an entry condition – begin the transaction only if the amount of subscribers enlisted exceeds a

given minimum. The census phase will fail if the conditions are not satisfied within a given timeout period. Conceptually, HTS supports any condition as long as the condition is based on data available to the publisher.

2.3.1.2 Transaction Phase

After successfully completing the census phase, a transaction is established, its participants are notified and the transaction phase begins. In the transaction phase, the publisher may publish many events within the transaction's context that will be handled by the middleware as an atomic unit. At any point, the publisher or enlisted subscribers may request that the entire transaction be aborted. The transaction phase ends when the publisher requests the middleware to commit the transaction.

2.3.1.3 Commitment Phase

The goal of the commit phase is to atomically apply all events within the transaction at the publisher and at all enlisted subscribers. To achieve this goal the 2PC (two-phase commit) protocol is used. The first phase of 2PC is a voting mechanism in which HTS sends a 'prepare' message to all enlisted subscribers and they vote to either abort or commit the transaction. If any subscriber votes to abort, the transaction fails and is aborted. If all subscribers vote to commit, HTS moves on to the second phase of 2PC in which a 'commit' message is sent. At this point all subscribers are responsible for committing the transaction and freeing their locked resources.

2.3.2 HTS Architecture

HTS supports two architectures:

1. *Client side* – HTS relies exclusively on a pub/sub middleware, and
2. *Service side* – HTS itself is a middleware. In this paper we refer to the service side architecture that has been implemented by the HTS team and is closer to the architecture we propose for TOPS.

HTS implements a transaction context by giving each transaction a transaction ID (txID) when the transaction is created. This ID is used for all messages within the transaction including publications and transaction messages use internally by HTS such as 'abort' and 'commit'. All clients participating in the transaction receive the txID during the census phase. HTS's txIDs are allocated using a Universally Unique Identifier (UUIDs) algorithm. A UUID is a 128 bit number that can be easily created and is guaranteed to be globally unique and a very high probability. A UUID is generated using the information such as: a host ID (e.g. MAC address or domain name), a time tag a random number. Several types of UUIDs exist, HTS utilizes Java's API for UUID creation which is implemented using the Leach-Salz algorithm documented in RFC 4122.

The middleware's transaction service is implemented by placing a transaction manager (tx manager) at each client (Figure 2-C) The tx manager provides the interface between the transaction and the application, and it coordinates distributed transactions with other tx managers.

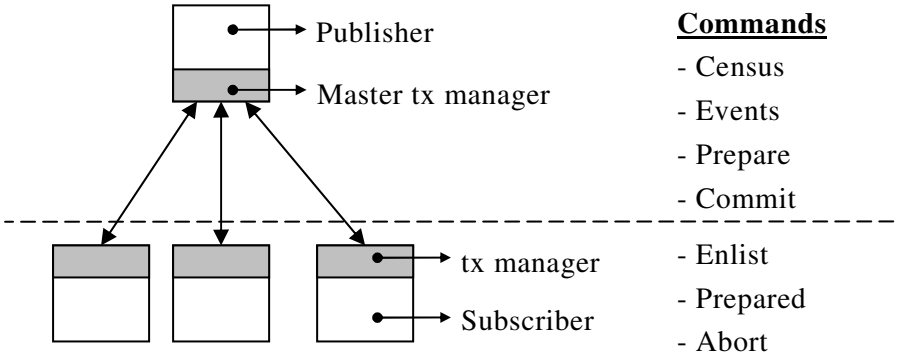


Figure 2-C: HTS Transaction Management

The tx managers utilize the txID to filter out messages belonging to transactions the client is not participating in. The tx managers manage the current transaction phase, of the three transaction phases listed above. In the census phase, the master tx manager is responsible for defining the txID, collecting responses from subscribers, and sending a message to participating subscribers containing a list of hashed subscriber IDs thereby ending the census phase. In the transaction phase the tx manager forwards the transaction's publications to the subscribers. In the commitment phase the tx manager initiates and manages the 2PC. Transactions are managed by the publisher's tx manager that is defined to be the *master tx manager*. This tx manager has the additional responsibility of coordinating among all participating tx managers. For instance, if one tx manager aborts the transaction, the master tx manager is responsible for instructing all other tx managers to abort.

2.3.3 Transaction Context

When publishing events of various types within the same transaction, all the events must be received by the participating subscribers and must be associated to the same transaction. HTS publishes events to a transaction in the same manner the events would have been published if they had not been part of a transaction. In other words, each event is published with its own event-type. In effect, this means that each event may be routed though a different rendezvous node. In order for the subscriber to receive all events belonging to the transaction, he must subscribe to all event-types that the transaction uses. If any participant failed to subscribe to a single event-type, he will be missing events belonging to the transaction and, therefore, the transaction will constantly be aborted.

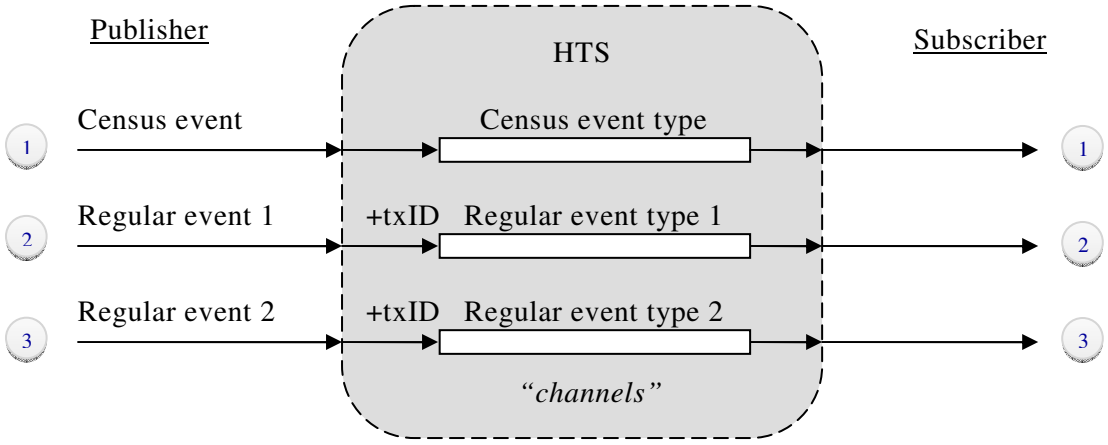


Figure 2-D: HTS Transaction Context

In Figure 2-D the publisher initiates a transaction using the census event 'Census event' of type 'Census event type' and publishes two other events within the transaction 'Regular event 1' and 'Regular event 2' of types 'Regular event type 1' and 'Regular event type 2' respectively. In such a case, the subscriber is required to subscribe to three event-types in order for the transaction to succeed: 'Census event type', 'Regular event type 1', and 'Regular event type 2'.

When publishing events to a transaction, HTS at the publisher's end appends a transaction ID to the event. Using this transaction ID the subscriber are capable of reconstructing the transactions context by grouping events with the same ID. This ID is also used by the subscriber when replying to the publisher, for instance during census or during commit.

2.3.4 Additional HTS Features

This section briefly describes particular HTS features relevant to this work.

- *Public / private transactions:* During the census phase of a transaction, the group of participating subscribers is determined. Only these subscribers are part of the transaction. Still, subscribers not participating in the transaction are capable of receiving the published events. If the transaction is private, only participating subscribers will receive the transaction's events. The events are filtered out by the subscribers' tx manager. If the transaction is public, all subscribers receive the events, including subscribers not enlisted in the transaction. Subscribers not enlisted in the transaction may fail to receive an event or fail to handle the event even though the transaction committed

successfully. On the other hand they may receive and handle an event belonging to a transaction that has been aborted.

- *Compensatable / non-compesatable clients:* A compensatable client is capable of retroactively rolling back a transaction even after it has already been committed. Compensation is difficult and not always possible; however, when the capability exists it can be utilized to shorten the commitment phase. The purpose of the commitment phase is to provide atomic commitment of all events within the transaction. The 2PC protocol is used to ensure atomic commitment for non-compensatable clients, for which there is no turning back after commitment. Compensatable clients, however, may commit immediately without waiting for the 2PC, knowing that in case of a rollback the committed changes can be undone. Avoiding 2PC improves performance by shortening delays and freeing locked resources sooner.

2.4 Introduction to Database Replication

Distributed databases are comprised of many database servers. Clients of the database may query different servers for the same data. It is the responsibility of the database management system to propagate changes between the servers in order to provide users at any server with the impression that the server he is accessing contains a complete and up-to-date database. Replication is commonly used to achieve this goal. Replication [23] is the process of propagating changes between servers efficiently while maintaining data integrity.

Replication may also be used as a backup mechanism. All changes to a primary server are replicated to a backup server. The backup server becomes active only in case of failure at the primary server. With replication in place, the backup server is continuously updated and it is, therefore, capable of replacing the primary server.

The two most basic types of replication are synchronous (eager) replication and asynchronous (lazy) replication. In synchronous replication changes to all replicas are performed atomically within a transaction. All replicas must be updated before the transaction can commit. Synchronous replication suffers from long locks on many servers and long delays caused by the two-phase commit (2PC) protocol used to commit transactions atomically. In asynchronous replication updates are committed locally and then replicated to other replicas without any dependencies between them. In this manner, commitment is performed quickly and locks are held for only a short period of time. A problematic situation occurs when the changes fail on any of the replicas causing the database to remain in an unstable state.

One possible application of pub/sub is for implementing database replication. In distributed databases, replication[1] must be used in order for changes at one copy to be propagated to other copies. Using synchronous replication all clients accessing the DBMS will always receive the same information, regardless of which copy they query. This method requires a mechanism providing distributed transactions, which inevitably causes long delays due to exclusive locking of the remote copies. Using asynchronous replication data is propagated between copies over time thereby allowing inconsistencies between sites. In spite of this disadvantage, most commercial DBMSs implement asynchronous replication because the performances issues stated above make synchronous replication somewhat impractical [2], [15].

Replication is further discussed in Chapter 5.

2.5 Related Work

2.5.1 Pub/sub Transactions

The work most related to this paper is the work done on HTS [26] which is extensively described throughout this thesis. The TOPS middleware supports transactions more inherently than HTS does. The resulting differences are pointed out in section 4.1. In addition to HTS, work relevant to this paper includes SPOONBILL [13], MMT (Middleware Mediated Transactions) [9] and two prototypes: Dependency-Sphere [25] and X^2TS [10], that realize MMT. Dependency-Sphere is a transactional middleware which allows capabilities of MOM and OOM (Object Oriented Middleware) to be encompassed within a single transactional context. Messaging in Dependency-Sphere includes queue messaging but not pub/sub messaging. X^2TS support pub/sub; however, it is only topic-based and does not support content-based pub/sub. In [13], a framework named SPOONBILL, which complements X^2TS is presented. SPOONBILL integrates transactions into content-base pub/sub. The SPOONBILL prototype uses Siena [3] as the underlying pub/sub middleware.

A primary difference between HTS/TOPS and the middleware presented above is content-based routing. The only other middleware that supports content-based routing is SPOONBILL, which is somewhat similar to HTS. One main difference between HTS and SPOONBILL is the census phase. In HTS the census phase is conducted at transaction initiation and determines the group of subscribers participating in the transaction. If

[1] Replication may also refer to mirroring- copying all data to a select set of backup sites for redundancy. Pub/sub may also be used in this case; however several of its strengths (such as scalability and anonymity) may not be utilized.

census conditions fail, no transaction is opened and no messages are sent. SPOONBILL performs its census at commit time allowing more flexibility in handling subscribers that drop out. Nonetheless, since census is at the end of the transaction, many more resources are wasted until it is discovered that census fails. Another key difference in the favor of HTS over SPOONBILL is that HTS is type-safe. This feature is derived from the underlying pub/sub middleware- HTS is base on Hermes that provides type-checking, SPOONBILL is based on Siena which does not. The importance of type safety is presented in [5] where a high level abstraction of type-based pub/sub (TPS) is introduced. [5] also includes a survey of pub/sub history and an overview of many pub/sub systems. SPOONBILL supports three possible visibilities: *immediate*, *differed* and *on-commit*, however, it lacks support for the new features we present in this paper, such as local transactions, access and scope.

2.5.2 Database Replication using Pub/Sub

Current databases such as SQL-Server [14], Oracle [17] and DB2 [7] each provide their own replication solutions, some of which incorporate pub/sub concepts. Of the three, only Oracle supports synchronous replication. SQL-Server supports *snapshot replication* that creates a onetime copy of data, *merge replication* which is *asynchronous peer-to-peer*, and *transactional replication* which is *asynchronous primary site* (no global transaction context). DB2 supports asynchronous multi-master replication; however, not synchronous replication.

Some suggest that replication be part of the middleware [18]. Domaschka et al. [4] argue that replication should be a layer above the middleware and not part of the middleware. They suggest an architecture to decouple replication and the middleware. TOPS qualifies as the latter since it includes no replication specific capabilities. From TOPS' point of view, the replication layer is an application.

In order to benefit from the performance of asynchronous replication and still achieve serialization, more advanced protocols (such as [2]) combine elements of both synchronous and asynchronous dissemination methods. For such protocols, a middleware offering support for both synchronous and asynchronous replication is likely to be even more beneficial than it is for the fundamental protocols we discuss in Section 5.1. [27] further addresses and emphasizes the benefits of a combined synchronous and asynchronous mechanism. They implemented the mechanism at the low level of the operating system/file system. This low level solution is irrelevant to our high level pub/sub solution; however, their argument for combining synchronous and asynchronous mechanism still remains.

3 Analysis of HTS – The Application Simulator

As any middleware, HTS is intended to serve applications requiring the services provided. In order to evaluate the functionality and performance of HTS, an application simulator was developed within the scope of this thesis. This simulator serves as a tool towards designing a new middleware, TOPS, introduced in section 4, which is based on HTS. This simulator was required in order to investigate the behavior of HTS and thereby acquire a deep understanding of its capabilities and potential drawbacks. It is important to note that the experience of developing over HTS was no less important than using the application simulator once complete. The simulator was used to help understand how HTS does what it is capable of doing, whereas the development experience raised ideas of what could be done differently than HTS or what could be done beyond the capabilities of HTS. The outcome of this process is detailed is section 4.1.

Being a middleware, HTS cannot be used standalone. It provides functionality to applications running above it. Applications have different needs and therefore will each use the middleware in a different manner. The challenge of the application simulator is to allow an operator to access and respond to the middleware in all possible legitimate manners. By exposing all the middleware functionality to the operator, the operator has the ability to simulate the behavior of any application.

In section 3.1, the requirements from the application simulator are identified and analyzed. The requirements analysis is performed by the means of use cases in accordance with the UML standard.

3.1 Requirements & Requirement Analysis

The simulator is required to allow the user to perform several types of actions in order to fully simulate a pub/sub environment. These actions are divided into several phases of the execution. In the *object definition phase* the environment is set up. Publishers and subscribers are defined as well as event-types. Next is the *pub/sub Setup phase* in which publishers advertise event types (census & regular) and subscribers register (subscribe) to event types (census & regular). In the *census phase* the publisher initiates a transaction (by publishing a census event) and subscribers choose whether or not they enlist in the transaction. Finally, the *transaction phase* is reached if the census phase concludes successfully. In the transaction phase, regular (not census) events are

published and the transaction is committed. Clients may choose to abort the transaction as well. The following list summarizes the capabilities listed above per phase:

Object definition phase:

- Define Publisher clients.
- Define Subscriber clients.
- Define event-types.

Pub/sub Setup phase:

- Advertise (Publisher)
- Register (Subscriber)

Census phase:

- Publish census event (Publisher)
- Enlist in transactions (Subscriber)

Transaction phase:

- Publish events (Publisher)
- Abort transaction (Enlisted Subscriber or Publisher)
- Commit transaction (Publisher)

3.1.1 Requirements Analysis: Use-Case Diagram

From the perspective of the simulator, there is only one actor which is the user. However, since the user is actually playing several roles, each role is considered to be an actor. The following three roles are defined:

- Manager – define objects (publishers, subscribers, event-types) needed to run a simulation.
- Publisher – a transaction initiator and event generator.
- Subscriber – a transaction participant.

Actions internal to the pub/sub system, such as setting up encryption keys and creating event brokers, are not listed since they are performed transparently to the user.

Actor: Manager

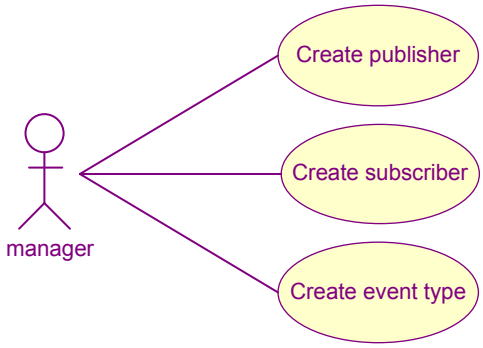


Figure 3-A: Use-Case Diagram (Manager)

Actors: Publisher, Subscriber

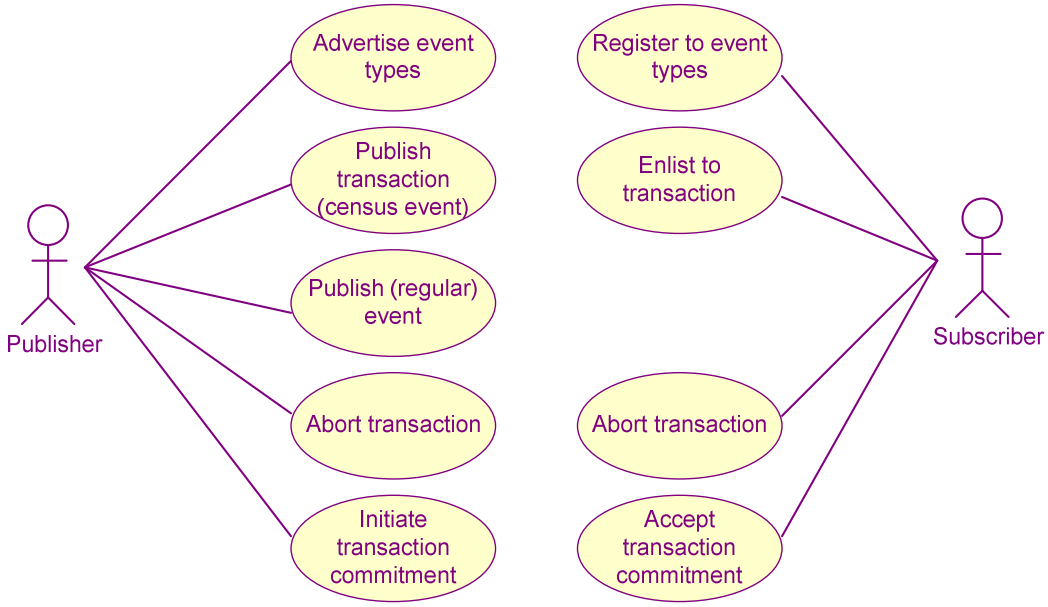


Figure 3-B: Use-Case Diagram (Publisher and Subscriber)

The following subsections contain several use-cases. More were defined but only a few main use-cases, which provide an understanding of pub/sub interaction, are included here. Note that the actions listed are actions taken by the user of the application simulator. Actions automatically performed by the middleware in response to user actions are not listed as actions; rather they appear as the outcome of the user's action.

For clarity of the use-cases, note that the interface between the application and HTS is based on API function calls. Notifications to the application initiated by HTS are implemented using callback functions (a function residing in the application is passed to HTS allowing HTS to call the application's function).

3.1.2 Use-Case 1: Successful Transaction

Actors:

Manager, Pub01, Sub01, Sub02.

Use-Case Description:

Publisher Pub01 initiates a transaction consisting of the census event type ETCensus01, and two other event types ET01 and ET02. Subscribers Sub01 and Sub02 enlist in the transaction. Pub01 sends two events, one event of type ET01 and one of type ET02. The subscribers accept them successfully. Pub01 requests commitment of the transaction, Sub01 and Sub02 accept and the transaction is committed successfully.

Phase	Actor	Action	Action effect
Object definition	Manager	create Pub01	Client Pub01 is created and added to P/S system.
	Manager	create Sub01	Client Sub01 is created and added to P/S system.
	Manager	create Sub02	Client Sub02 is created and added to P/S system.
	Manager	create ETCensus01	Create a census event type named ETCensus01.
	Manager	create ET01	Create an event type named ET01.
	Manager	create ET02	Create an event type named ET02.
P/S Setup	Pub01	advertise ETCensus01	Event type is stored in rendezvous node.
	Pub01	advertise ET01	Event type is stored in rendezvous node.
	Pub01	advertise ET02	Event type is stored in rendezvous node.
	Sub01	register to ETCensus01	Subscriber registers to event type.
	Sub02	register to ETCensus01	Subscriber registers to event type.
Census	Pub01	publish ETCensus01	Send a notification of ETCensus01 publication to subscribers.
	Sub01, Sub02	enlist to transaction	Census callback function will return 'true'- a message will be sent to publisher, and subscriber will automatically be subscribed to all event types in transaction registration.
Transaction	Pub01	publish ET01	Send an event of type ET01 to subscribers.
	Sub01, Sub02	nothing	Event is accepted if no exception is thrown.
	Pub01	publish ET02	Send an event of type ET02 to subscribers.
	Sub01, Sub02	nothing	Event is accepted if no exception is thrown.

	Pub01	initiate commit	Sends an internal 'prepare' event to subscribers (2PC).
	Sub01, Sub02	accept commit	User confirms preparation for commit. After all clients confirm preparation, the publisher sends the internal 'commit' event upon which the subscribers commit.

3.1.3 Use-Case 2: Census Not Fulfilled

Actors:

Manager, Pub01, Sub01, Sub02.

Use-Case Description:

Publisher Pub01 initiates a transaction consisting of the census event type ETCensus01, and the event type ET01. At publication of the census event, the publisher sets the minimum amount of participants to 2. Only subscriber Sub01 enlists to the transaction. Once the census phase at the publisher times-out, the census phase will fail due to lack of participants. No transaction is created.

Phase	Actor	Action	Action effect
Object definition	Manager	create Pub01	Client Pub01 is created and added to P/S system.
	Manager	create Sub01	Client Sub01 is created and added to P/S system.
	Manager	create Sub02	Client Sub02 is created and added to P/S system.
	Manager	create ETCensus01	Create a census event type named ETCensus01.
	Manager	create ET01	Create an event type named ET01.
P/S Setup	Pub01	advertise ETCensus01	Event type is stored in rendezvous node.
	Pub01	advertise ET01	Event type is stored in rendezvous node.
	Sub01	register to ETCensus01	Subscriber registers to event type.
	Sub02	register to ETCensus01	Subscriber registers to event type.
Census	Pub01	publish ETCensus01 Require a minimum of 2 subscribers	Send a notification of ETCensus01 publication to subscribers.
	Sub01,	enlist to transaction	Census callback function will return 'true'- a message will be sent to publisher, and subscriber will automatically be subscribed to all event types in

			transaction registration.
	Sub02	nothing	Since only one subscriber enlisted to the transaction, the census at the publisher will fail after a timeout. No transaction will be created.
Transaction	-	-	-

3.1.4 Use-Case 3: Commit Fails

Actors:

Manager, Pub01, Sub01, Sub02.

Use-Case Description:

Publisher Pub01 initiates a transaction consisting of the census event type ETCensus01, and the event type ET01. Subscribers Sub01 and Sub02 enlist to the transaction. Pub01 sends one event of type ET01. The subscribers accept the events successfully. Pub01 requests commitment of the transaction, Sub01 responds positively but Sub02 fails to commit and sends an abort message to the publisher. The transaction in aborted.

Phase	Actor	Action	Action effect
Object definition	Manager	create Pub01	Client Pub01 is created and added to P/S system.
	Manager	create Sub01	Client Sub01 is created and added to P/S system.
	Manager	create Sub02	Client Sub02 is created and added to P/S system.
	Manager	create ETCensus01	Create a census event type named ETCensus01.
	Manager	create ET01	Create an event type named ET01.
P/S Setup	Pub01	advertise ETCensus01	Event type is stored in rendezvous node.
	Pub01	advertise ET01	Event type is stored in rendezvous node.
	Sub01	register to ETCensus01	Subscriber registers to event type.
	Sub02	register to ETCensus01	Subscriber registers to event type.
Census	Pub01	publish ETCensus01	Send a notification of ETCensus01 publication to subscribers.
	Sub01, Sub02	enlist to transaction	Census callback function will return 'true'- a message will be sent to publisher, and subscriber

			will automatically be subscribed to all event types in transaction registration.
Transaction	Pub01	publish ET01	Send an event of type ET01 to subscribers.
	Sub01, Sub02	nothing	Event is accepted if no exception is thrown.
	Pub01	initiate commit	Sends an internal 'prepare' event to subscribers (2PC).
	Sub01	accept commit	User confirms preparation for commitment.
	Sub02	abort	Preparation for commit failed. Send 'abort' to publisher and roll back. Publisher will send an 'abort' to Sub01 and will rollback.

3.1.5 Use-Case 4: Multiple Successful Transactions

Actors:

Manager, Client01, Client02.

Use-Case Description:

Client Client01 initiates a transaction consisting of the census event type ETCensus01, and the event type ET01. Client Client02 who is registered to ETCensus01, enlists to the transaction. At the same time, Client02 initiates a transaction consisting of the census event type ETCensus02, and the event type ET02. Client Client01 who is registered to ETCensus02, enlists to the transaction.

Client01 sends one event of type ET01 which is accepted by Client02. Client02 sends one event of type ET02 which is accepted by Client01. Each publisher requests commitment of its transaction and accepts commitment of the others transaction. Both transactions are committed successfully.

Phase	Actor	Action	Action effect
Object definition	Manager	create Client01	Client Client01is created and added to P/S system.
	Manager	create Client02	Client Client02is created and added to P/S system.
	Manager	create ETCensus01	Create a census event type named ETCensus01.

	Manager	create ETCensus02	Create a census event type named ETCensus02.
	Manager	create ET01	Create an event type named ET01.
	Manager	create ET02	Create an event type named ET02.
P/S Setup	Client01	advertise ETCensus01	Event type is stored in rendezvous node.
	Client01	advertise ET01	Event type is stored in rendezvous node.
	Client02	register to ETCensus01	Subscriber registers to event type.
	Client02	advertise ETCensus02	Event type is stored in rendezvous node.
	Client02	advertise ET02	Event type is stored in rendezvous node.
	Client01	register to ETCensus02	Subscriber registers to event type.
Census	Client01	publish ETCensus01	Send a notification of ETCensus01 publication to subscribers.
	Client02	enlist to transaction	Census callback function will return 'true'- a message will be sent to publisher, and subscriber will automatically be subscribed to all event types in transaction registration.
	Client02	publish ETCensus02	Send a notification of ETCensus02 publication to subscribers.
	Client01	enlist to transaction	Census callback function will return 'true'- a message will be sent to publisher, and subscriber will automatically be subscribed to all event types in transaction registration.
Transaction	Client01	publish ET01	Send an event of type ET01 to subscribers.
	Client02	publish ET02	Send an event of type ET02 to subscribers.
	Client01	initiate commit	Sends an internal 'prepare' event to subscribers (2PC).
	Client02	accept commit	User confirms preparation for commit. After all clients confirm preparation, the publisher sends the internal 'commit' event upon which the subscribers commit.
	Client02	initiate commit	Sends an internal 'prepare' event to subscribers (2PC).
	Client01	accept commit	User confirms preparation for commit. After all clients confirm preparation, the publisher sends the internal 'commit' event upon which the subscribers commit.

3.1.6 Interaction with HTS and API Requirements

It is an elementary requirement that the simulator know to interact with HTS. This interaction must be at the level of the API as well as the behavioral level which are both

analyzed in detail in the sequence diagram on page 29. In the diagram, the publisher is on the left side and the subscriber is on the right. Only one subscriber is shown, however, there may be many. In the center, the publisher and subscriber meet within the underlying pub/sub system. The internal behavior of the pub/sub system is not of interest and simply grayed out. Over the pub/sub layer, HTS runs on both the publisher's side and the subscriber's in order to provide transactional capabilities. Even though HTS is part of the middleware, its behavior is shown since it is vital for understanding how the simulator must behave. Above HTS is the simulation application that runs two separate entities, one to simulate the publisher and another to simulate the subscriber. The following sections detail the required interaction in each phase.

Object Definition Phase

This phase is not shown in the sequence diagram due to space limitations. The simulator begins by setting up the network over which the middleware will run. The default network consists of five event brokers. Next, the simulator allows the user to setup the environment of the simulation including clients and event types. Clients are logged on to the middleware by validating their credentials. Only clients with authorized keys are successfully logged on. The simulator automatically uses valid key for all clients. The event types the user defines are not passed to the middleware at this point. The middleware will become aware of the event types only once the event type is advertised. The simulator has the user enter the event types already at this point in order to provide the user with a convenient selection list of event types at all clients. Note that even though the object selection phase must come first, new objects may be defined at any time during the following phases.

Pub/Sub Setup Phase

In this phase, advertisements and registrations are performed. A publisher interested in publishing an event of a certain type must first advertise its intention of doing so. Event-types may be advertised as census events, events used to initiate transactions, or regular events, events which are sent within a transaction. A subscriber interested in being notified when a transactional event of a certain type is published, must first register to the appropriate event types. This registration includes the type of the census event of the transaction and the types of regular events that may be received within the transaction. HTS will subscribe the subscriber to the census event immediately, but will subscribe to the contained regular events only once a transaction is successfully joined. The subscriber must additionally provide HTS with a callback function that is used by HTS to notify the subscriber of an incoming transaction. This callback is further discussed in the census phase.

Census Phase

The census phase begins when a publisher initiates a transaction. The publisher requests the HTS middleware to publish an event of a census event type. This request must include conditions under which the census phase is to be considered successful (e.g. at least 2 subscribers). The middleware now calls the census callbacks (given during registration) of all subscribers. These callbacks are the way the middleware notifies the application of initiation of a transaction. The choice of joining the transaction is left to the subscriber's application. If the subscriber's callback returns 'false', the middleware will not join the transaction. If the callback returns 'true' the middleware will join the transaction by subscribing to all necessary event types and by sending the publisher a reply to its publication.

The census phase is limited to a period of time defined by the initiating publisher. Once this timeout elapses the transaction will either be in 'census fulfilled' state or 'census unfulfilled' state. It is the responsibility of the publisher to poll the middleware and to instruct it to begin the transaction if the census phase is fulfilled. In either case, the census phase ends here.

Transaction Phase

The transaction phase is the phase in which work of interest to the application is accomplished. The publisher publishes events within the transaction and the middleware makes sure all subscribers receive it. HTS calls the event callback at each participating subscriber in which the subscriber takes actions of interest according to the received event. If the event is handled successfully, the callback must return 'true'. If the event is not handled successfully, the callback must return 'false' in which case the middleware marks the transaction as aborted.

The publisher publishes as many events as it likes and eventually decides to commit the transaction. The HTS middleware initiates the two-phase-commit by sending a 'prepare' command to all participants of the transaction. If any participant replies with the 'abort' message the publisher aborts the entire transaction by sending 'abort' to all participating subscribers. If all participants reply to the 'prepare' command with the 'prepared' message the publisher commits the transaction and sends the 'commit' command to all participants. At this point the transaction is successfully completed. The publisher will also abort the transaction if within a timeout period a participant failed to respond to the 'prepare' command.

The simulator must poll HTS to find out the status of the transaction.

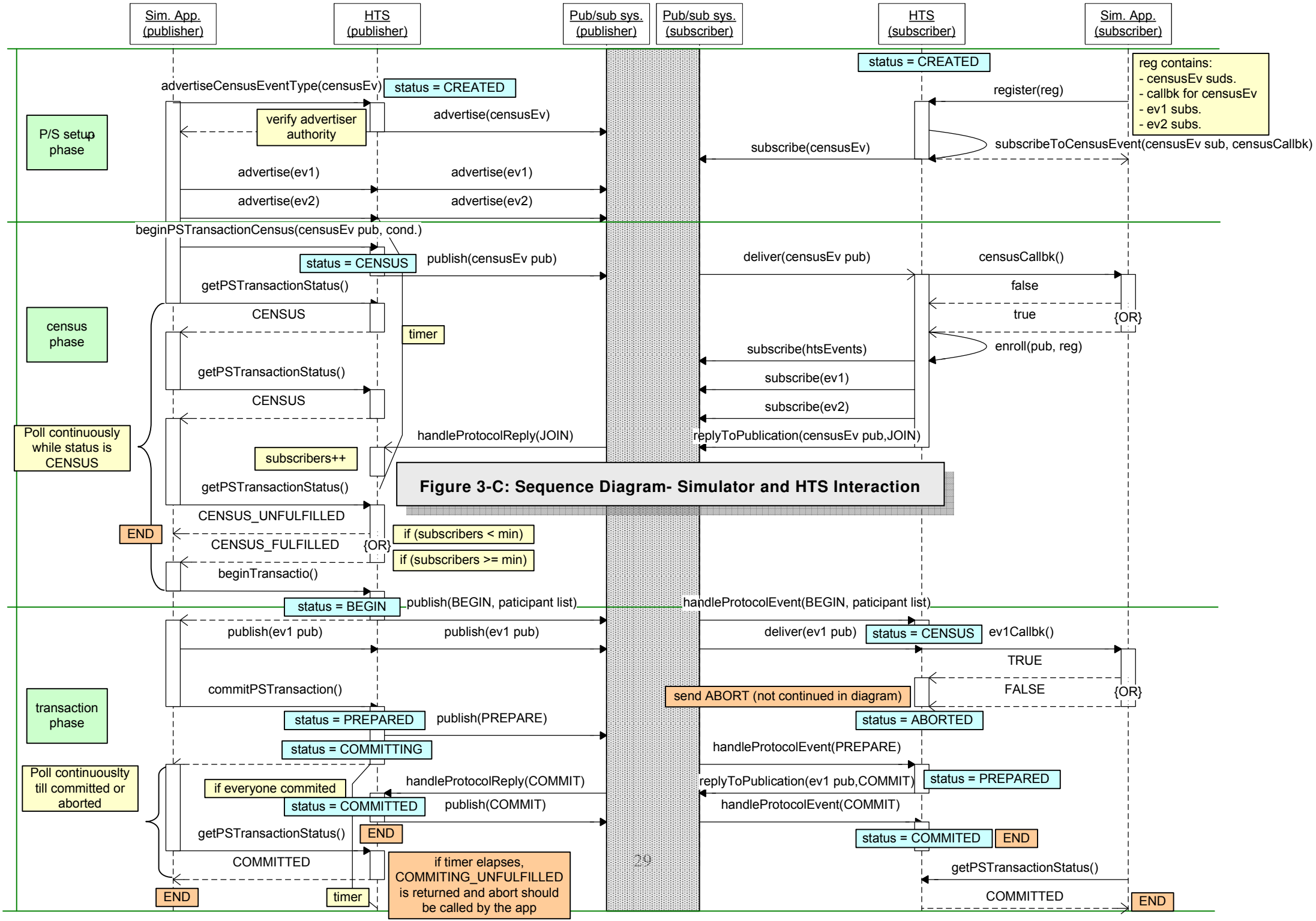


Figure 3-C: Sequence Diagram- Simulator and HTS Interaction

3.2 Simulator Design

3.2.1 Software layers

Figure 3-D shows the different layers the application simulator and the middleware are comprised of. Layers above the horizontal line belong to the application and layers below the horizontal line belong to the middleware. The layers on top are high level software modules and the layers on the bottom are low level software modules.

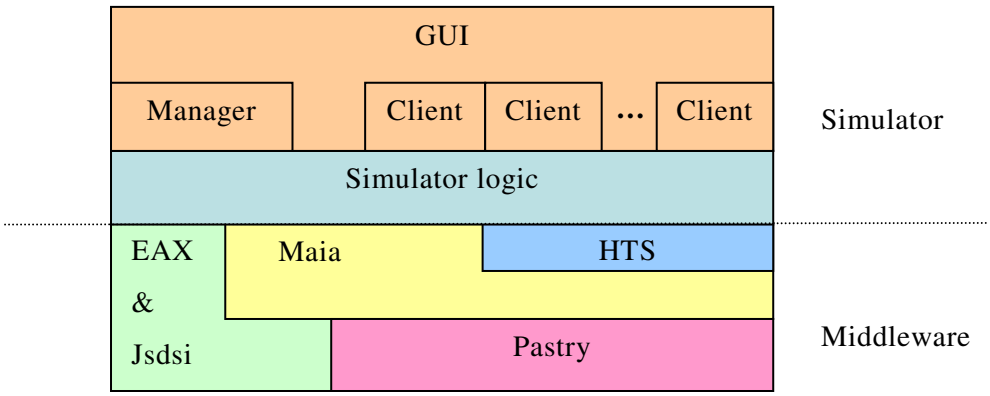


Figure 3-D: Software Layers

Layers of the Simulator

The simulator is composed of two layers: the GUI layer and the simulator's logic layer.

The GUI (Graphical User Interface) provides the user with necessary windows for setting up the pub/sub system and using it to publish events and subscribe to events. The GUI layer may be replaced with another GUI without influencing the logic and behavior of the simulator. The GUI contains two types of windows: a manager window and a client window.

- Manager window– A window used by the Manager actor to setup the environments. Allows defining clients and event-types.
- Client window– A window representing a client (publisher or subscriber). This window allows the client to advertise event-types (at the publisher side), register to event-types (at the subscriber side), publish events (at the publisher side) and subscribe to events (at the subscriber side).

Only one manager window may exist, whereas many client windows may exist simultaneously.

The simulator logic layer contains all the simulator's data and functionality. Only this layer accesses the middleware. It contains the existing event-types and clients and allows the GUI to query their status. The GUI layer, accesses the logic layer in which all the data it displays is stored. The logic layer also handles user events generated by the GUI and notifies the GUI of changes caused by the middleware. At startup, this layer initializes the pub/sub environment (e.g. network nodes and event brokers) and allocates encryption keys for all clients. In order for objects to be identifiable by a user, objects such as clients are given a textual name. The name is mapped to an object and translated when going out to, or coming in from the user. The GUI and middleware are both unaware of this translation.

Layers of the Middleware

The middleware is composed of several components:

- HTS – Hermes Transaction Service provides transactional functionality to the Hermes pub/sub middleware. HTS is the primary layer that the simulator accesses.
- Maia - The implementation of the Hermes pub/sub middleware, including the security and transaction services.
- Pastry - The peer-to-peer routing substrate used by the Hermes routing system.
- EAX & Jsdsi - Jsdsi (Java Secure Distributed Software Infrastructure) is used by Maia to enforce access control via SPKI. EAX offers cryptographic extensions used by Maia as well.

Source code for all the middleware's layers were provided to us by Cambridge University for the purpose of this research.

Interaction between Layers

Each layer accesses services provided by the layers directly beneath it, and may invoke callbacks belonging to the layer above it.

The GUI layer, accesses the simulator logic layer via an interface, and provides the simulator logic layer with an interface to an object that handles events (incoming event etc.). Thanks to the interfaces, the GUI and logic are completely decoupled allowing one to be replaced with the other remaining unaffected.

The simulator logic layer accesses HTS to provide it with transactions, Maia to setup and run the pub/sub system, and the EAX & Jsdsi layer to achieve encryption keys and services. These layers are created by the simulator logic layer and their objects are

accessed directly (they do not provide interfaces). The simulator logic defines and provides HTS with the callbacks necessary to handle transactions (refer to section 3.1.6).

The interaction between HTS and Maia were detailed in section 3.1.6. The interactions between the remaining layers within the middleware are of no consequence to this project and are therefore ignored.

3.2.2 Class Diagrams

As described in section 3.2.1 the simulator is comprised of two layers: the GUI later and the simulator logic layer. Separation of the two layers is achieved by using interfaces. The GUI creates an instance of the simulator logic layer it is to run above and from this point on all communication is through interfaces.

Figure 3-E is a UML class diagram which contains the main classes and their most important member function and variables. In this diagram, a solid line with a closed hollow head arrow depicts inheritance, a dashed line with an open head arrow depicts a dependency, and a solid line with a solid diamond on one end and an open head arrow on the other end depicts composition.

The application begins by running the main window (*MainWindow* class). *MainWindow* creates a single instance of *LinkToApp* class. *LinkToApp* contains an object of the *HtsSimApp* class which contains the simulators logic. *LinkToApp* provides the GUI with an interface (*IHtsSimApp*) to the functionality of the logic layer.

Through the main window, the user has the capability of creating clients. For each client created, *MainWindow* creates a *ClientWindow* object and calls createNewClient() in *IHtsSimApp* which creates a new object of *ClientWrapper*. Additionally, *createNewClient()* in *IHtsSimApp* must be called to give *ClientWrapper* access to the callback functions of *IGuiClientCallback*.

HTS and the other layers below the simulator are created and accessed by the *HtsSimApp* class. *HtsSimApp* contains a list of publishers, subscribers, and event-types. It is not sufficient to rely on lists managed by HTS since they are not identifiable by the user. These entries in *HtsSimApp* lists connect each entry in the HTS lists with additional data such as a user identifiable unique name or history log. The publisher and subscriber lists in *HtsSimApp* contain objects of the *ClientWrapper* class which provides additional required data as described above. The *ClientWrapper* class also

contains HTS callbacks that all clients require. These callbacks activate the GUI callbacks in *IGuiClienCallback* when necessary.

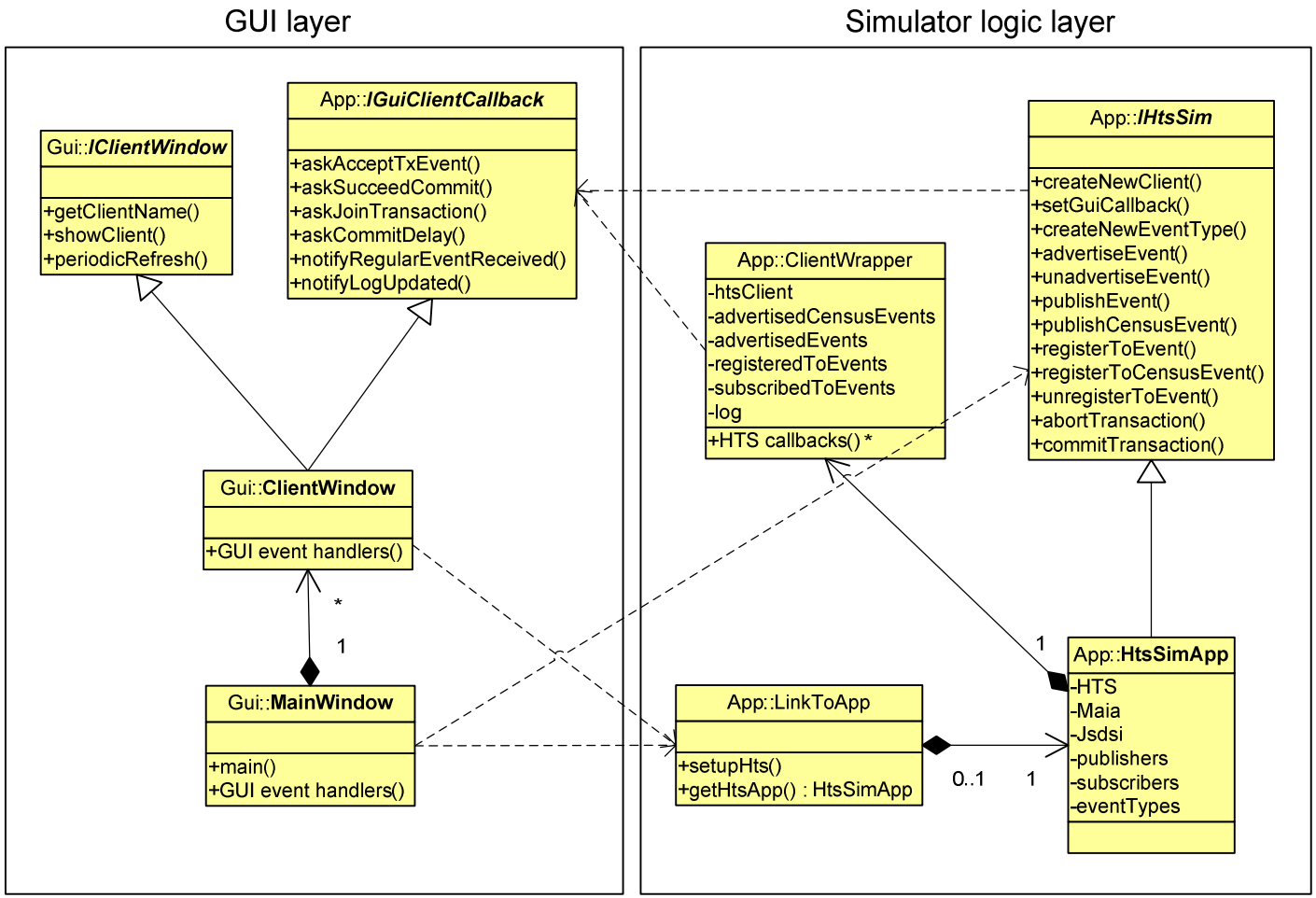


Figure 3-E: Application Simulator Class Diagram

3.2.3 Threads

The main thread of the application is the *main()* function of the *MainWindow* class. For each client, a client window (*ClientWindow*) is created. Client windows are modeless and are each a thread. In addition to the main thread, *MainWindow* contains a timer thread that is responsible for refreshing the display of the main window and all client windows. Once instructed to, the windows refresh themselves by rereading data from the logic layers using the *IHtsSimApp* interface.

Calls from the GUI to the logic layer are all run in the context of the GUI thread (main thread or client window thread). The middleware (HTS or layer below) contain threads that handle incoming events. These events activate callback in the logic layer, which, in turn, may activate callbacks of the GUI layer. All the callbacks are run in the context of the middleware.

The logic layer contains one thread that is responsible to poll HTS after a census event is published in order to obtain the outcome of the census phase. This thread is necessary because HTS does not provide callback support for this event.

4 TOPS – Transaction Oriented Publish/Subscribe

In this chapter we present the design of a transactional pub/sub middleware that improves upon HTS and provides additional capabilities as will be described throughout this chapter. The motivation for transactional messaging is traditionally illustrated using the 'meeting scenario' [9], [26], [13]. In this scenario, a meeting is scheduled by sending invitational messages to potential attendees. Participation of an attendee may be either optional or mandatory. Attendees reply to the meeting initiator to confirm their participation. In addition, a room must be reserved for the meeting. It is desirable that the room reservation and the updates to all attendees' calendars take place only if the mandatory attendees confirmed attendance, the minimum quota of attendees it met and the room is available for use. This requirement for a single unit-of-work may be achieved using a messaging middleware supporting transactions, and specifically to our interest, a pub/sub messaging middleware. This scenario may be slightly artificial but it serves well as an example and therefore we use it to demonstrate the motivation for TOPS in Section 4.1.

We introduce data replication as another application of transactional pub/sub. In the replication example, an update is performed to data residing on a particular site and the change is to be reflected, synchronously or asynchronously, at all other sites containing replicas of the modified data. This is discussed in chapter 5.

4.1 Motivation for TOPS

The main goal of TOPS is to provide a diversity of features for supporting transactions in a pub/sub environment. TOPS does not attempt to provide a single ultimate solution, rather it attempts to provide a variety of features, allowing applications to determine the most appropriate solution (or set of solutions) base on the application's requirements. The benefits of TOPS listed below are demonstrated in chapter 5 by presenting how a variety of replication strategies may be implemented using TOPS.

Even though HTS is a good starting point and we base many of our concepts on it, several design and architectural changes are necessary to achieve our goals. Following are several of the reasons an improved design is necessary.

4.1.1 Transactions with Multiple Publishers

In HTS a transaction may only have one publisher. Using the meeting scenario, suppose one secretary notifies of the meeting's existence and another determines and notifies of the meeting location. Each secretary is a separate publisher, but still each must publish an event within this transaction. TOPS allows any number of publishers to contribute events to a transaction.

Sample scenario

Publisher 'A' (Figure 4-A) initiates a transaction and publishes an event containing the date and time of a meeting. Publisher 'B' publishes another event within the transaction notifying all participants of the meeting's location. Both events (time and location) are encompassed within the same transaction even though they originate from different publishers.

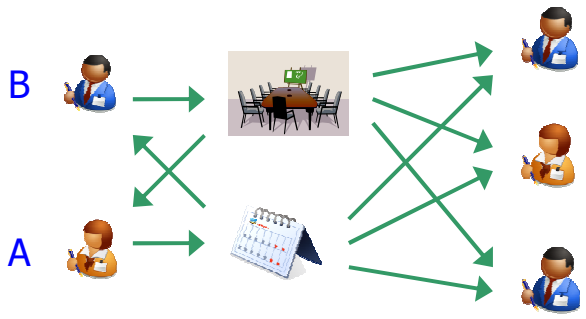


Figure 4-A: Transactions with Multiple Publishers.

4.1.2 Transaction Initiator is not Required to Participate

In HTS the initiator must participate in the transaction. It is not possible for one to schedule a meeting of behalf of a colleague. The scheduling of the meeting will fail if the scheduler's schedule is not free at the time of the meeting, even though all invited attendees have accepted the meeting. In TOPS, the initiator is not required to be part of the transaction.

Sample scenario

Publisher ‘A’ initiates a transaction on behalf of ‘B’ and publishes an event containing the date and time of a meeting. Publisher ‘A’ himself is not part of the meeting transaction and is not required to attend (he may not even be a subscriber). The transaction scope is denoted in Figure 4-B by a broken line. ‘B’ may still join the meeting transaction as an ordinary subscriber

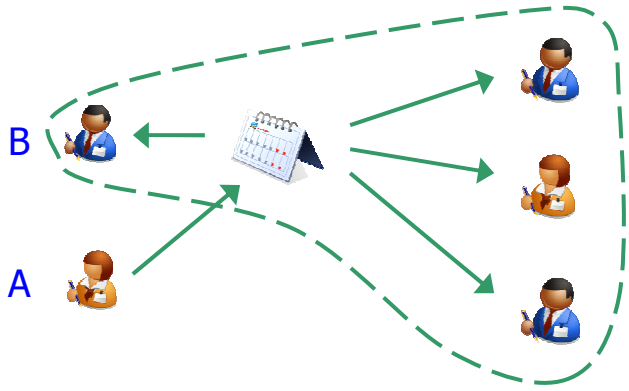


Figure 4-B: Transactions with Initiator not Participating.

4.1.3 No Need to Explicitly Subscribe to Transactions

In HTS all participating subscribers must explicitly subscribe to transactions (census events). Suppose the pub/sub middleware contains the event type ‘staff meetings’. In HTS in order to support a transaction, a new census event type, say ‘staff meetings census’ must be defined, which all potential attendees must subscribe to as well. TOPS allows the publication of transactional and non-transactional events within the same event type. The single event type ‘staff meetings’ is sufficient and attendees must subscribe only to it.

Sample scenario

The subscribers (right hand of Figure 4-C) subscribe to a single event type ‘staff meeting’. ‘A’ publishes a standard event of this type (not in a transaction) and it is received by all subscribers. ‘B’, on the other hand, publishes an event of the same type but he does so within a transaction. His transactional pub/sub event is received by all subscribers as well, as part of a transaction. Distinguishing between transactional events and non-transactional events at the subscriber is handled by the TOPS middleware.

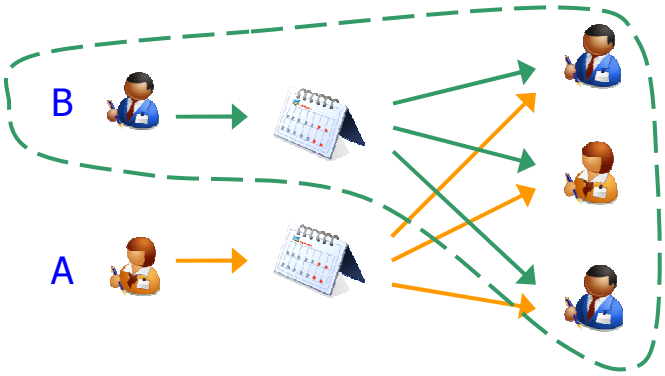


Figure 4-C: No Need to Explicitly Subscribe to Transactions.

4.1.4 Automatic Delivery of Any Event-Type

In HTS all participating subscribers must subscribe to all event types used in the transaction. Assume a meeting transaction consists of an event of type 'meeting schedule' and an event of type 'meeting location'. For attendees to meaningfully participate in the transaction they must receive both events. In HTS this is achieved by the subscriber listing, during subscription to the census event, all the event types that may be published within the transaction. This is a relatively inflexible solution since event types may not be easily added to a transaction. TOPS requires only a single subscription. If a transaction is created with event type 'meeting schedule', participants in the transaction will receive a 'meeting location' event published within the transaction context without having subscribed to 'meeting location'. To see how this is achieved, see section 4.5.3.

Sample scenario

The subscribers have subscribed to the event type 'meeting schedule' only. Publisher 'A' initiates a transaction of type 'meeting schedule' and the subscribers join the meeting transaction. 'A' now publishes an event of type 'meeting location' within the transaction. All the meeting's participants will receive this event as well, even though they haven't subscribed to this event type.

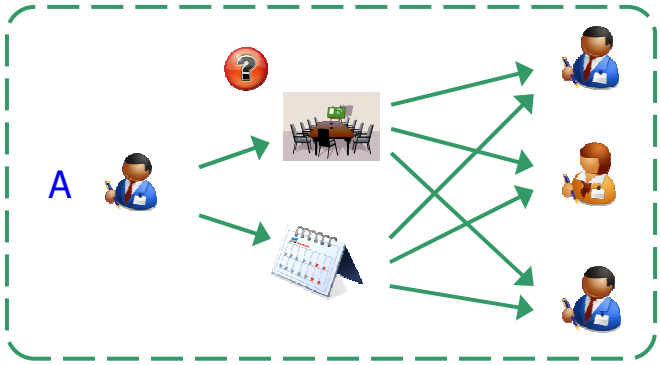


Figure 4-D: Automatic Delivery of Any Event-Type.

4.1.5 Test of Constraints at Commit

In HTS constraints are tested only during transaction establishment (the census phase). Assume a meeting has been defined with the constraint that the meeting shall take place only if there are at least twenty attendees. Assume also that the census phase was successfully concluded with twenty five attendees, but during the transaction one attendee aborted and rolled back. Since a part of the transaction failed, HTS will abort the transaction even though the constraint requiring a minimum of twenty participants is still fulfilled. TOPS finalizes the list of participants before committing, allowing five attendees to abort before canceling the meeting.

Sample scenario

A meeting is scheduled using a transaction with a constraint requiring participation of at least two attendees. The three subscriberes shown in Figure 4-E joined the transaction, however the top subscriber failed to commit the transaction. Since contraints are tested at commit, the transaction will successfully commit with the two remaining subsacribers.

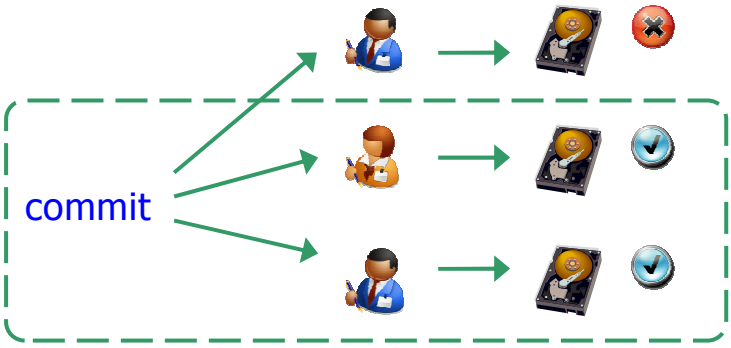


Figure 4-E: Test of Constraints at Commit.

4.1.6 Support for Local Transactions

In HTS only distributed transactions are supported. Suppose a conference is scheduled in one of an organization's two distant offices. One event schedules the conference and another designates a conference room in the desired office. We require a transaction to support the case in which it does not matter who accepts the invitation, yet we would like to ensure that either both the schedule and location are accepted or that neither events are accepted. On one hand, atomic execution at each subscriber is desirable, which calls for a transaction. On the other hand the outcome of one subscriber should not affect the outcome of any other subscriber, thereby preventing use of a distributed transaction. Local transactions solve this problem by having each participant run a transaction individually. The different types of transactions are further discussed in the following section (4.2).

Sample scenario

A meeting transaction composed of two events, a time event (E1) and a location event (E2) in scheduled. Three subscribers join the transaction and receive both events. One atendee (the top atendee in Figure 4-F) failed to commit the location event and therefore the time event will not be applied either. Since each atendee is running its own local transaction, only the failed atendee aborts while the rest of them succcessfully commit the meeting transaction.

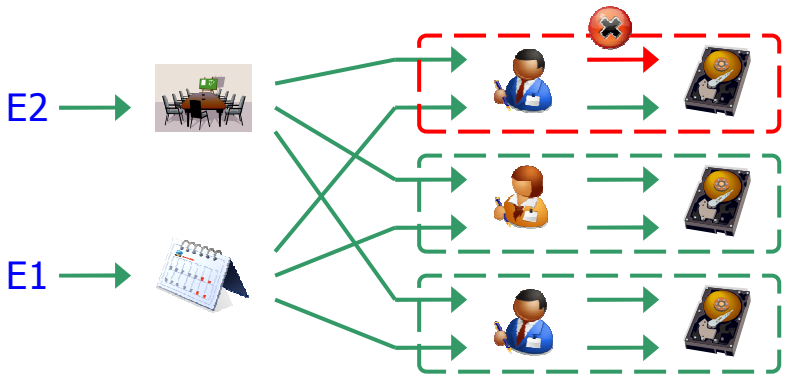


Figure 4-F: Support for Local Transactions.

4.1.7 Motivation Conclusion

The points listed above and their scenarios gave the motivation for a new design for a transactional pub/sub middleware. In the following sections, we describe the new design (TOPS) in detail.

4.2 Transactions Supported

As with a non-transactional middleware, the application may take upon itself the responsibility of transaction management. In this case, transactions are managed above the TOPS middleware, and the middleware is responsible only for distributing events. Such transactions, of course, do not require a transactional middleware and do not utilize its capabilities. The main advantage of a transactional middleware is freeing the application of transaction management.

We distinguish between two types of transactions that middleware can manage (Figure 4-G): *local transactions* and *distributed transactions*.

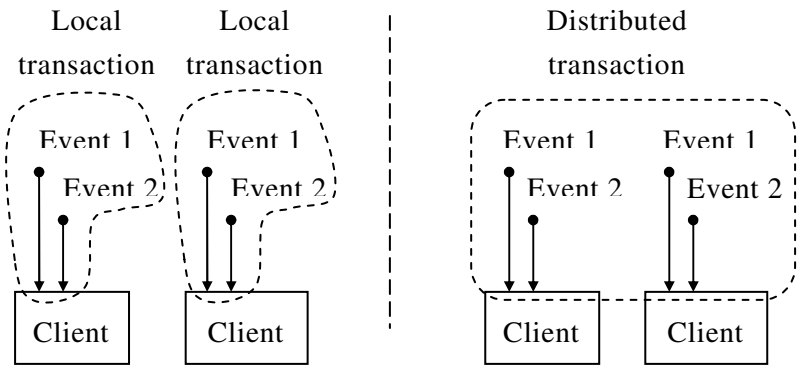


Figure 4-G: Local transactions vs. distributed transactions.

The type of a transaction is determined by the *scope* concept (see Section 4.4). Note that for the same events, it is possible that some subscribers receive the events as a distributed transaction, others as local transactions, and yet others as events not within a transaction at all.

4.2.1 Local Transactions

Local transactions are executed at a single site. Even though the transaction is limited to a single site, it may be necessary to duplicate the transaction to additional sites, each executing the transaction locally (no dependencies between sites exist). We refer to this type of transactions as *local transactions* because the transaction spans only a single site. For example, an initiating site executing events as a local transaction may be interested in asynchronously passing on the events to remote sites as a unit of execution. Even though he is indifferent to which remote sites will receive or apply the events, he is interested in forcing atomic execution of the events within a local transaction at each remote site in order to preserve data integrity. In chapter 5 it is

shown how certain types of database replication can be implemented using local transactions.

For local transactions TOPS provides a transactional context at each site and provides the dissemination of the transaction's events to each site. The initiating site publishes each step (event) of the transaction to the middleware in a transactional context. The middleware is fully responsible for distribution and execution of the transaction as a local transaction on each remote site. Each site's local transaction commits or aborts individually with no dependencies between sites (including the initiating site). Furthermore, the publisher itself may prefer having the middleware manage its own transaction as well.

An example scenario for the use of local transactions was given in Section 4.1.6.

4.2.2 Distributed Transactions

Distributed transactions involve transactions at many sites, each executing a sub-transaction. Each sub-transaction is, in effect, a local transaction since it involves only a single site. In a distributed transaction, all sites participating in the transaction depend on the outcome of the others. If one site fails, all sites must abort (4.4.4 covers some situations in which this requirement is slightly relaxed). Unlike local transactions in which many transactions are executed, a distributed transaction is a single transaction that encompasses all participating sites. Each site individually executes a sub-transaction. TOPS provides a local transactional context for sub-transactions at each site, and a distributed transaction context encompassing all sub-transactions in to a single distributed transaction.

TOPS supports distributed transactions by providing a local transactional context for sub-transactions at each site, and a global transaction context that atomically combines all sub-transactions. TOPS also handles dissemination of the transaction's events to participating sites. The publishing site(s) determine what events the transaction contains and when it will be committed. All other participating sites affect the outcome of the transaction only by failing, thereby causing the distributed transaction to abort.

4.2.3 Mixing Distributed and Local Transactions

Since the ability of determining the transaction type is given to the subscribers, it is possible for the same published transaction to have some clients that are executing the transaction as a distributed transaction while others are executing the transaction as a local transaction. This is calleda mixed transaction. Subscribers executing the transaction as a distributed transaction do not affect other subscribers executing the

transaction locally, and subscribers executing the transaction locally do not affect other local or distributed transaction subscribers.

4.3 Event Dissemination

The event dissemination scheme follows that of Hermes. Each event type is routed to a rendezvous node. Different event types may have different rendezvous nodes. Within an event type, any event published will reach the same rendezvous node. The rendezvous node is the root of a multicast tree, which is constructed during clients' subscriptions and is used for event dissemination.

In addition to the transaction's events, distributed transactions involve communication between sites for management of the transactions, for example, the events belonging to the commitment protocol. Even though this traffic occurs within the TOPS middleware and is transparent to the application, it should be taken into account that due to this traffic, a bandwidth and latency overhead exists when using transactions.

In a simple algorithm for event dissemination, the publisher sends its events directly to the rendezvous node and only then dissemination begins. Some algorithms use a slight optimization that allows dissemination to begin before the event reaches the rendezvous node. This situation occurs when a node residing on the path from the publisher to the rendezvous node is part of the multicast tree. In such a case, this node can immediately disseminate the events to its sub-tree concurrently to forwarding the events to the rendezvous node.

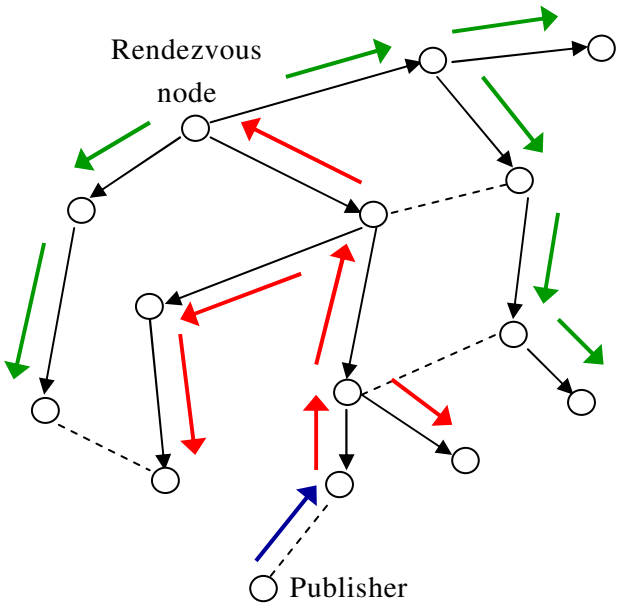


Figure 4-H: Optimization of Rendezvous Node Multicast.

Figure 4-H illustrates this situation. The arrows directly between nodes denote a physical connection and the direction of communication within the multicast tree. Dotted lines denote a physical connection not part of the multicast tree. Arrows alongside the graph denote the path used for event dissemination by the labeled publisher. In this example, the rendezvous node's left and right sub-trees receive the event from him but the entire sub-tree in the center did not receive the event from him.

This is relevant to TOPS in order to support multiple publishers. Transactions with multiple publishers use the rendezvous node to order events from all publishers and therefore dissemination optimizations as described in this section must be disabled. For single publisher transactions the optimization may remain in place.

4.4 Transaction Properties

4.4.1 Participation of Initiator

In order to allow for flexibility as to having the transaction initiator participate, TOPS does not automatically enlist the initiator in the transaction. If the initiator is interested in participating, he is required to join the transaction during the census phase, just as any other subscriber. In addition to not requiring the initiator to participate, we also gain the ability of having many publishers in a transaction.

4.4.2 Access

Access to a transaction defines who may publish events within the transaction. In a transaction of *single publisher (private)* access, only the initiating site may be a publisher, whereas transactions of *multi-publisher (public)* access may have several publishers. In transactions with many publishers the content of the transaction is distributed as well as the outcome of the transaction. The transaction initiator determines the transaction's access.

4.4.3 Scope

At each subscriber, when a new transaction arrives, the subscriber determines the scope this transaction will have from his point of view. The scope of a transaction refers to the ability of subscribers to view events within a transaction. A subscriber may request the following levels of scope:

1. *Get none* – This subscriber will not receive events belonging to this transaction.
2. *Get* as *non-transactional* – This subscriber will receive events but they will not be processed as a transaction.

3. *Get* as *local* – This subscriber will receive events within a local transaction.
4. *Get* as *distributed* – This subscriber will receive events within a distributed transaction.

4.4.3.1 Scope Limitation

The transaction initiator has the ability to limit the scope that subscribers may request. For example, if the initiator limits the scope to 'distributed', subscribers may only choose to join the distributed transaction, or to not join at all. The initiator may restrict the scope by limiting the transaction to:

1. *Limit none* – No restriction. Subscribers may request any scope.
2. *Limit local* – Events may be provided only within a local or a distributed transaction. Subscribers may request either 'Get as local' or 'Get as distributed'.
3. *Limit distributed* – Events may be provided only within a distributed transaction. Subscribers may request only 'Get as distributed'.

The 'scope' and 'scope limitation' may not be broader than the transaction type. For instance, if the transaction type is 'local', the scope and scope limitation may not be 'get as distributed' or 'limit distributed' respectively. Furthermore the scope must be higher than the scope limitation. For instance, the limitation is 'limit local' the subscriber may not request a scope of 'get as non-transactional'.

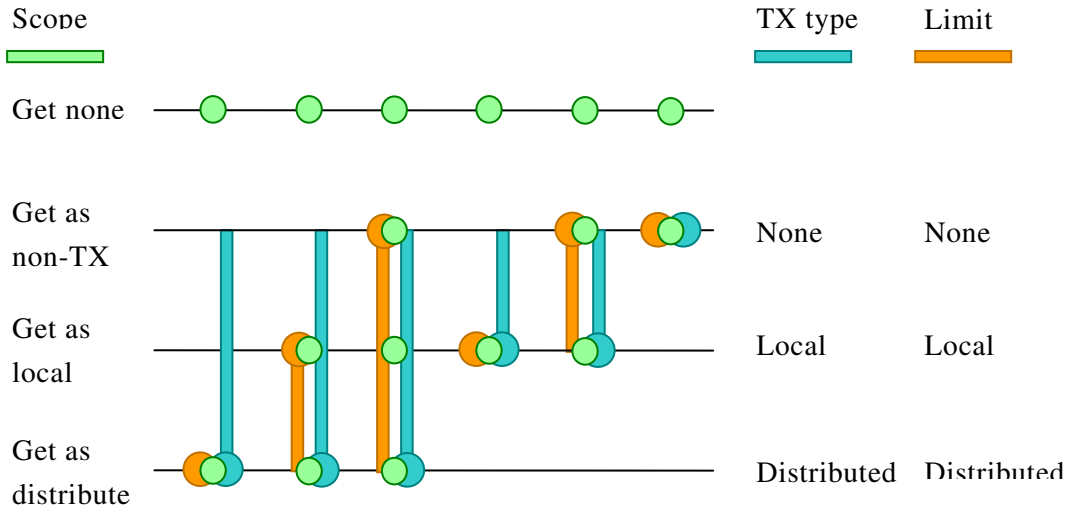


Figure 4-I: Legal Scopes According to Transaction Type and Scope Limitation.

In Figure 4-I it is shown what scope limitations are possible for each transaction type, and what the scopes are possible given the transaction type (TX type) and limitation. First, the transaction type is defined. A blue circle is placed at the appropriate strip and a bar is drawn upwards. This bar determines the possible scope limitations. Next, the limitation is defined. An orange circle is placed at the appropriate strip and a bar is drawn downwards until the TX type's circle. Legal scopes, which are denoted by a slightly smaller green circle, are between the TX type's circle and the limitation's circle. The scope 'get none' is exceptional and is always a legal scope.

The scope concept is an extension to private/public transactions in HTS. 'Limit none' is similar to 'public' in HTS, and 'limit distributed' is similar to 'private' in HTS. Differences between TOPS and HTS still exist due to the fact that HTS does not support local transactions. A demonstration of how access and scope are useful is given in Section 5.2 where we discuss replication. Table 4-A summarizes what the subscriber will receive for all combinations of scope request and limitation. Of course, the limitation may not be higher that the transaction itself (e.g. limit distributed is illegal for a local transaction).

Table 4-A: Transaction Scope

	limit none	**limit local**	**limit dist.**
get none	none	none	none
get as non-tx	non-tx	none	none
get as local	local	local	none
get as dist.	dist.	dist.	dist.

Rows – Scope requested by a subscriber.

Columns – Scope limitation as defined by the initiator.

4.4.4 Constraints

Constraints may be applied to transaction in which case the transaction must fulfill all given constraints in order to commit. Examples of constraints are: require a minimum number of participants, require participation of specific subscribers, limit participation to specific subscribers, completion within a defined time, etc. These constraints are in addition to constraints external to the middleware, such as database constraints. The constraints are tested throughout the life of the transaction (census and commitment) and invalid transactions are aborted. Most importantly, constraints are tested prior to commitment to ensure validity of committed transactions.

In HTS [26] census of participants and transaction constraints are conducted at transaction creation. This solution prevents non-interested clients from receiving the events belonging to the transaction. On the downside, constraint will not work as expected in the following case. Assume a distributed transaction contains a constraint requiring participation of 2 subscribers. Further assume that 3 subscribers joined the transaction. At commit, if one subscriber aborts, the entire transaction aborts, as expected from a distributed transaction. However, the constraint clearly allows transactions with 2 participants and therefore a legal transaction was completely aborted.

In SPOONBILL [13] census of participants and transaction constraints (failure conditions) are conducted at transaction commitment. This solution prevents the problematic situation described in the preceding paragraph. Unfortunately, all subscribers (including ones not interested in the transaction) will receive all the transaction's events, and all the transaction's events are sent even if it is clear from the beginning that the transaction will be aborted (e.g. not enough participants).

TOPS attempts to benefit from both worlds by conducting the census phase up front, as in HTS, and retesting the constraints at commit as in SPOONBILL. In this manner, only interested subscribers receive events, transactions failing constrains up front are aborted without sending irrelevant events, and transactions will not fail at commit as long as the constraints are still met.

When referring to constraints regarding transaction participants, the consequence of retesting constrains at commit is not trivial. In HTS, once census is complete, the list of participants is finalized, and all participants are part of the distributed transaction. This is the default behavior in TOPS as well, when no constrains are specified. However, once participation constraints are retested at commit, we didn't really have a true distributed transaction until now, as we see some participants can abort and the transaction still commits with the remaining participants. In TOPS, a more accurate description of this situation might be "applying a global constraint to local transactions". In effect, such a transaction remains a distributed transaction, perhaps just of a more generic type, since a constraint is placed when the transaction is created that defines when a transaction involving many sites is allowed to commit. Even though aborted site may be removed from the transaction, it remains impossible for a site will to commit when the transaction is abort.

4.5 TOPS Architecture

TOPS is a single middleware that provides two interfaces: a standard pub/sub interface and a transactional pub/sub interface. The standard pub/sub interface is exposed by the *P/S manager* and the transactional pub/sub interface is exposed by the *tx manager*. The P/S manager's interface is generic; however, its implementation is specific to TOPS.

TOPS is run over a communication substrate, as in HTS. Transactions and applications may access the communication substrate directly, bypassing pub/sub, thereby enabling point-to-point communication. For example, the tx manager of a subscriber may send an 'abort' message directly to the coordinating tx manager. This architecture is depicted in Figure 4-J.

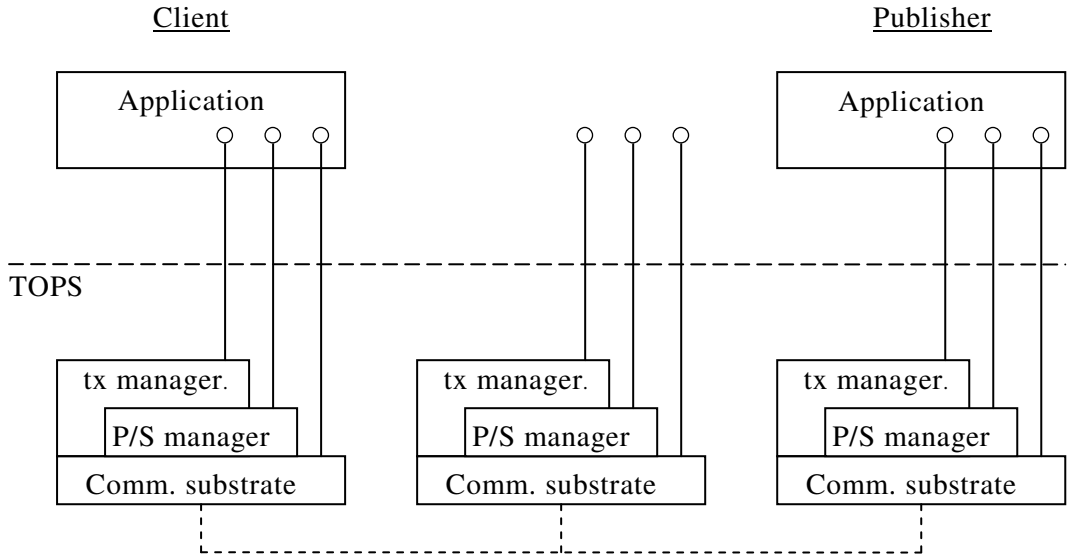


Figure 4-J: TOPS Architecture

4.5.1 Transaction Management

The tx manager is fully responsible for all aspects of the transaction including: census to determine participants, adhering to constraints, maintaining a global transaction context and distributing events within it, aborting transactions, and committing transactions. The tx manager is more of a logical module, since it physically spans many sites including: publishers, the rendezvous node, and all subscribers. Having identified three types of sites (publisher, rendezvous node, and subscriber), we divide the tx manager into three physical components, the *event coordinator*, the *tx coordinator* and the *sub-tx executor*. Before going into the responsibilities of each

component, it is important to note the rationale behind this division. If the tx manager was a single component, clearly the component will be performing different responsibilities (and therefore executing different code) at each type of the sites. One part will be active at the publisher whereas a completely different part will be active at the subscriber. Dividing into components allows each component to be completely active or completely inactive. For instance, a site will never use half the functionality of the event coordinator. This architecture provides a high level of flexibility as follows: All components are deployed at all sites. Depending on the transaction, different components at different sites cooperate in order to fulfill the requested transaction.

The key responsibilities of the three components are as follows:

1. *Event coordinator* – When an event is received requesting a new transaction, the event coordinator allocates a new transaction ID (txID). Events received may be sent from one or more publishers, the event coordinator determines which events published are to be part of the transaction and numbers them in sequence. The event coordinator sends the numbered events to the tx coordinator(s).
2. *Tx coordinator* – The tx coordinator is responsible for distributed execution of a transaction. It coordinates transaction establishment, event publication, and transaction commitment by communicating with all (one or more) sub-tx executers. The tx coordinator coordinates between all sub-transactions to form a single distributed unit-of-execution.
3. *Sub-tx executor* – Each subscriber participating in the transaction is managed by a local sub-transaction executor that provides a single local unit-of-execution. The sub-tx executor receives commands from the tx coordinator and executes them locally by interfacing with the application. For example, during transaction establishment a sub-tx executor will ask the application if it is interested in participating in the transaction, or it may send the tx coordinator an abort request when it fails to pass an event to the application. The sub-tx executor is also responsible for knowing if the application is compensatable or not, and instructing the application to roll back events belonging to aborted transactions accordingly.

Configuration of components is discussed in detail in section 4.5.2, however in order to explain why three layers were chosen, we require an introduction to component configuration. From a communications point of view, the components are placed in layers, meaning the publisher's application communicates with the event coordinator that communicates with the tx coordinator that communicates with the sub-tx executor that communicates with the subscriber's application. Communication between components may be either point-to-point or pub/sub.

Knowing the responsibilities of each component and having an idea of how they interact, it remains to explain why three components are required. Starting from the end, the sub-tx executor is required in order to connect the middleware to the subscriber's application (i.e. delivering events). Having only this layer is not sufficient since we require support for distributed transactions. For distributed transactions a single, centralized transaction manager is required. The tx coordinator fulfills this role. However, since local transactions are supported as well, and for local transaction the tx coordinator resides at the subscriber's site (each site has its own tx coordinator), yet another (thin) layer is required above the tx coordinator. The tx coordinator is responsible for controlling the entry of events into the transaction (possibly from several publishers) and disseminating them to all tx coordinators. To demonstrate the necessity of the event coordinator, assume there is no event coordinator in a multi-publisher local transaction configuration. Sites P1 and P2 publish events E1 and E2 respectively, into the local transaction. Subscribers S1 and S2 are running the local transaction. It is very possible that S1 will execute the events in one order, say (E1, E2) and S2 in another order (E2, E1). Clearly this is not desirable and violates the principals of a transaction. To solve this, all publishers in a particular transaction send their events through the same event coordinator. The event coordinator determines the order in which the events are to be executed, and sends them to all subscribers numbered in sequence. In this manner it is guaranteed that all subscribers are running the same local transaction.

4.5.2 Component Configurations

The tx manager of each site contains all of the above components. For a specific transaction, different components are in use at different sites. For a specific transaction, all layers must be passed though, and in order; however, the layers are not all at the same site. The configuration of the components depends on the type of the transaction.

A transaction must include exactly one event coordinator, regardless of the transaction type, since it is the entry point to the transaction. Publishers communicate with the event coordinator when creating a transaction or when publishing an event to the transaction. In order to support multi-publisher transactions (transaction with many publishers), the event coordinator is located at the rendezvous node. Since the rendezvous node is the root of the multicast tree, the event coordinator is sure to receive events from all publishers. The transaction initiator must subscribe to the event type the transaction is created under, if it wishes to participate in the transaction. If multi-publisher transactions are not required the event coordinator may be located at the initiating publisher.

Tx coordination differs between transaction type. When using local transactions (see 4.2.1) each site coordinates its own transaction. This requires a tx coordinator at each subscriber, whose scope is the local site only. Communication between the event coordinator and the tx coordinators is by pub/sub. When using distributed transactions (see 4.2.2) there is a single centralized tx coordinator that is chosen to be at the same site as the event coordinator. In this case, communication between the two coordinators is direct.

A sub-tx executor exists at each subscriber and is connected to a single tx coordinator. When using local transactions, the tx coordinator is local as well, resulting in direct communication between them. With distributed transactions, the tx coordinator is remote. Events from the tx coordinator to the sub-tx executor are sent using pub/sub (since communication is one-to-many), and events in the reverse direction are point-to-point (since communication is one-to-one). The type of communication used between the different layers is illustrated in Figure 4-K.

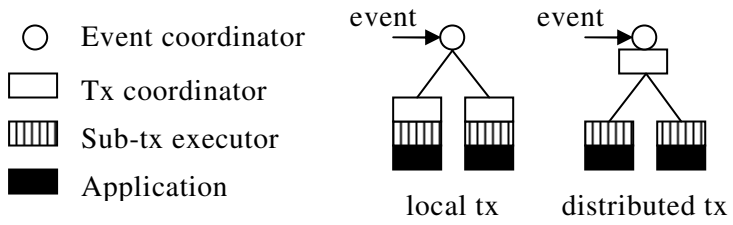
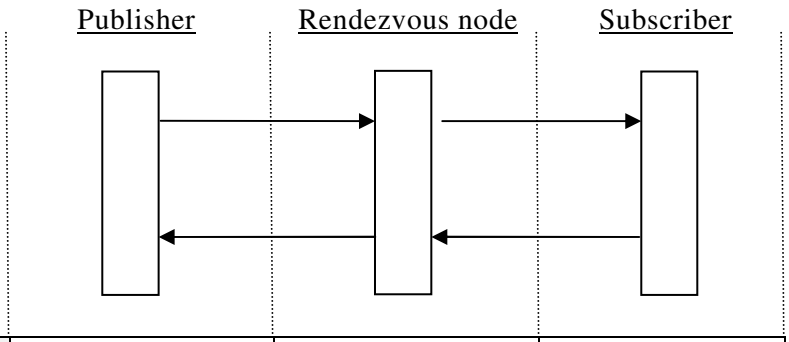


Figure 4-K: Communication in Tx Manager Configurations

Note that every TOPS node contains all the listed components, but each component is activated according to the currently active transaction scenario. Figure 4-K shows the configuration for only a single transaction. A node may concurrently serve more than one transaction in which case different components (or the same component) may serve different transactions. For instance, a site that is the initiator of an active, single publisher, distributed transaction (T1), and is the subscriber in an active local transaction (T2), will activate the components as follows: the event coordinator serves T1, the tx coordinator serves T1 and T2, and the sub-tx executer serves T2.

The configurations described above are summarized in Figure 4-L. For the various types of transactions, it is shown where each component resides.



	Transaction type	Publisher	Rendezvous node	Subscriber
1	Single-publisher & distributed	Event coordinator Tx coordinator		Sub-tx executor
2	Multi-publisher & distributed		Event coordinator Tx coordinator	Sub-tx executor
3	Single-publisher & local	Event coordinator		Tx coordinator Sub-tx executor
4	Multi-publisher & local		Event coordinator	Tx coordinator Sub-tx executor

Figure 4-L: Tx Manager Configurations

Sample Scenario

In Figure 4-M an actual scenario is given. In this scenario, two transactions with the same event-type are active.

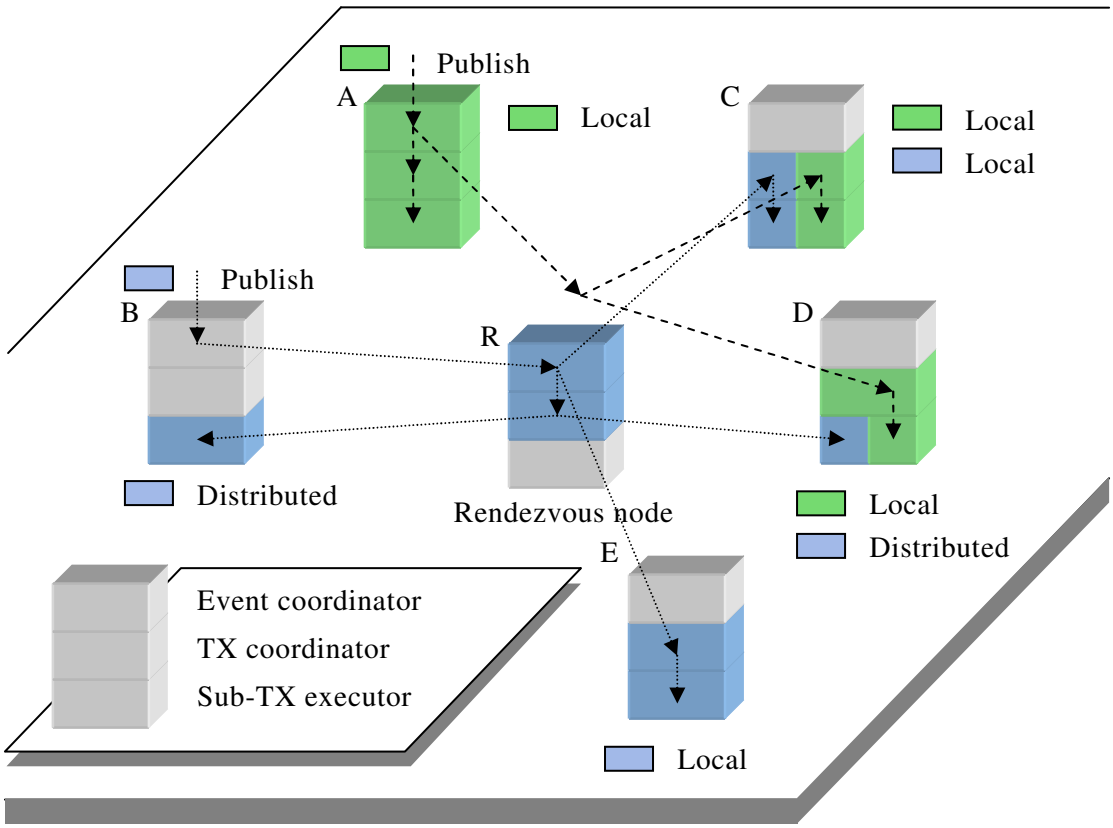


Figure 4-M: Component Configuration Scenario

Transaction T1 (green) was initiated by site A and was created as a local transaction with *single publisher* access. Because of the *single publisher* access, TOPS decided to place the event coordinator at the initiator's (A's) site. Sites C and D (in addition to A) joined the transaction, both with *local* scope. For local sites, each local site has its own TX manager. Sub-TX executors are always local. When A publishes an event (see figure), the event starts at the event coordinator located at A, next it is passed to A's TX manager directly and is also passed to all other subscribers' TX managers using pub/sub via the rendezvous node (site R). Finally, at each site, the event is passed down to the sub-TX executor and from there to the application.

Transaction T2 was initiated by site B. T2 was created as a distributed transaction with *multiple publisher* access. Because of the *multiple publisher* access, TOPS decided to place the event coordinator at the rendezvous node (R). Sites C, D and E (in addition to

B) joined the transaction, C and E with *local* scope, and B and D with *distributed* scope. Since local sites have their own TX manager, transaction T2 uses the TX coordinator at sites C and E. For the distributed clients B and D the TX coordinator at R is used. As before, sub-TX executors are always local. When B publishes an event (see figure), the event start at the event coordinator located at R. Next, it is passed to R's TX manager directly, and to C's and E's TX managers using pub/sub. Finally, C and E pass the event down to the sub-TX executor locally, and R uses pub/sub to pass the event down to B's and D's sub-TX executors.

From the point of view of a specific site, it can be seen that a single component can server many transactions. This occurs in the sub-TX executor component at C and D, and in the TX coordinator component at C. Additionally, a single component can server transactions of different types, as the sub-TX executor at D is serving both a local and a distributed transaction.

4.5.3 Event Sequencing

In order to insure that all subscribers of a transaction are executing the same exact transaction, TOPS gives each event within a transaction a transaction specific sequence number. The number is given by the event coordinator component. This sequencing is required in order to determine a single order of execution of events in a transaction with multiple publishers. Note that if the publishers are also participating in the transaction, they too receive the order of events from the event coordinator at the rendezvous node and do not rely on the order of events they published. Additionally, the sequencing is used by the tx-coordinator and the sub-tx executor in order to verify the integrity of the events. If a participant fails to follow the event sequence, it aborts the transaction.

4.5.4 Transaction Context

In Section 2.3.3 the transaction context of HTS was discussed. HTS publishes each event within a transaction to its own event-type. Due to the disadvantages involved in this method as stated above, TOPS publishes all events to the same event-type. In this manner, it is guaranteed that all events are routed through the same rendezvous node. The first advantage is that a transaction may have more than one publisher. Additionally, participants are only required to subscribe to one event-type, and are guaranteed that they will not miss any events due to lack of proper subscription. Yet another advantage is removing the need for census event-types altogether. Instead, transactions can be created using any event-type. When a publisher initiates a transaction, it must specify the event-type the transaction will be created under. TOPS will return the application a transaction ID (txID). Using this txID, the publisher can publish events of any type to the transaction. TOPS automatically associates the txID to the transaction's event-type and tunnels all events though the same event-type (and

rendezvous node). This method also allows a site to be the initiator of several transactions concurrently, a capability not offered by HTS.

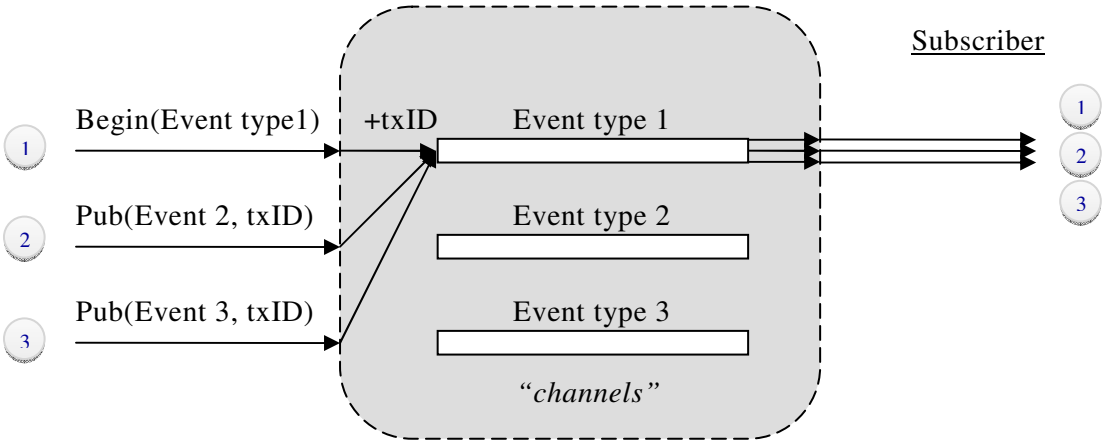


Figure 4-N: TOPS Transaction Context

Following Figure 4-N, suppose a publisher initiates a transaction using 'Event type1' and publishes two other events within the transaction 'event 2' and 'event 3' of types 'Event type 2' and 'Event type 3' respectively. Even though three different event-types are involved in the transaction, the subscriber is required to subscribe only to 'Event type 1' in order for the transaction to succeed. Through 'Event type 1' he will receive all the transaction's events.

Similarly to HTS, TOPS at the subscribers end is capable of reconstructing the transactions context by grouping events with the same txID.

4.6 Transaction Processing

As in HTS (see section 2.3), TOPS processes transactions in three phases:

1. Transaction establishment – census phase.
2. Transaction execution – transaction phase.
3. Transaction Commitment – commitment phase.

4.6.1 Transaction Establishment

When establishing a transaction, the initiator must supply the following information:

1. *Event-type* and content information under which the transaction will be created (all events in the transaction are wrapped by this event-type, see 4.6.2),
2. A *tx type* that may be either 'local' or 'distributed'.

3. *Access*- which is either ‘single publisher’ (private) or ‘multi-publisher’ (public),
4. *Scope limitation* – ‘none’, ’local’, or ’distributed’.
5. *Constraints* – to which the transaction must adhere.

According to the *tx type,* an event coordinator is determined, which then generates a txID and sends an establishment event to all tx coordinator(s). With local transactions, the establishment event is published and received by the tx coordinator of the subscribers to this event-type. Subscribers may join the transaction by enlisting at their local tx coordinator. No feedback is to be sent to the event coordinator, which is completely unaware of the tx coordinators. With distributed transactions, the event coordinator sends the establishment event directly to the single tx coordinator at the rendezvous node. The tx coordinator publishes the event to all subscribers. Subscribers interested in the transaction store the txID and the ‘get dist.’ scope (for use when receiving events belonging to this transaction), and respond to the tx coordinator with their (cryptographic) ID. The tx coordinator tests for fulfillment of the constraints and notifies the event coordinator if the establishment succeeded or failed. Specific subscribers may choose to receive the events with a different scope, either within a local transaction or simply as independent events. As before, these subscribers store the txID with the selected scope. In such cases, the subscriber silently listens to the publications of the tx coordinator and handles them as he pleases. Subscribers not at all interested in the events will not store the txID.

Once the constraints (see section 4.4.4) are fulfilled, all subscribers are notified and the transaction begins execution.

Algorithm pseudo code

Algorithm for the initiating publisher's logic:

```
BEGIN BeginTx(evType, txType, access, scopeLim, constraints[])

  // In a transaction with 'private' (single publisher) access the
  // 'event coordinator' is located at the initiator. In a transaction
  // with 'public' access the 'event coordinator' in located at the
  // rendezvous node.

  IF access = 'private' THEN
    Activate 'event coordinator' component locally (at publisher).
    Send internal 'begin TX' event to 'event coordinator'.
    Allocate txID.
    Store tx properties (evType, txType, access, scopeLim, constraints).
    Publish internal 'begin TX' event to 'tx coord.' including txID.
  ELSE
    Use 'event coordinator' component at rendezvous node.
    Publish internal 'begin TX' event to 'event coordinator'.
  END

  // If transaction is 'local' or no constrains are defined, transaction
  // is always successful. A distributed transaction with constraints
  // may fail and therefore must be tracked.

  IF txType = 'distributed' AND constraints are defined THEN
    Wait a timeout for 'tx coordinator' to acknowledge transaction.
    Wait for response.
    IF transaction successfully established THEN
      RETURN txID from response.
    ELSE
      RETURN transaction not established.
   END
END
```

Algorithm for the rendezvous node's logic:

```
BEGIN HandleBeginTx (evType, txType, access, scopeLim, constraints[])

  // Handle case of 'public' transaction when 'event coordinator' is
  // at rendezvous node.

  IF event is targeted at 'event coordinator' THEN
    Activate 'event coordinator' component.
    Allocate txID.
    Store tx properties (evType, txType, access, scopeLim, constraints).
    Publish internal event to 'tx coordinator' including txID.

    // Check if transaction is successful (same logic as at initiator).

    IF txType = 'distributed' AND constraints are defined THEN
      Wait a timeout for 'tx coordinator to acknowledge transaction.
      Wait for response.
     IF transaction successfully established THEN
       Send to initiator: txID from response.
     ELSE
       Send to initiator: transaction not established.
    END
  END

  // For distributed transactions, the tx coordinator is at the
  // rendezvous node. It is responsible for establishing the list
  // of participants and testing constraints.

  IF event is targeted at 'tx coordinator' THEN
    IF txType = 'local' THEN
      Publish event to 'tx coordinators'.
    ELSE // txType = 'distributed'
      Activate 'tx coordinator' component.
      Publish event to 'sub-tx executor'.
      Wait timeout for subscriber participation.
      Test constraints.
      tx status = fail.
      IF constraints pass THEN
        Store participants.
        tx status = ok.
      END
      Publish tx status to participants' 'sub-tx executor'.
      Send tx status to 'event coordinator'.
    END
  END
END
```

Algorithm for the subscriber's logic:

```
BEGIN HandleBeginTx(evType, txType, scopeLim)

  // For local transactions the 'tx coordinator' resides at the
  // subscriber and no feedback is required.

  IF txType = 'local' THEN
    Activate 'tx coordinator' component.
    Activate 'sub-tx executor' component.
    Acquire scope from application.
    SWITCH scope
      'get none': Do not stode txID.
      'get non-tx': Store txID with scope.
      'get local':
         Store txID with scope.
         Create a transaction.
    END

  // For distributed transactions the 'tx coordinator' resides at the
  // rendezvous node and must be notified when joining the transaction.

  ELSE // txType = 'distributed'
    Activate 'sub-tx executor' component.
    Acquire scope from application.
    SWITCH scope
      'get none': Do not stode txID.
      'get non-tx': Store txID with scope.
      'get local':
         Store txID with scope.
         Create a transaction.
      'get distributed':
         Store txID with scope.
         Send 'tx coordinator' at rendezvous node a request to join tx.
         Wait for tx status from 'tx coordinator' at rendezvous node.
         IF tx failed OR not joined THEN
           Delete stored txID.
         END
    END
  END
END
```

4.6.2 Transaction Execution

During transaction execution, events are published within the transaction's context with the help of the txID and are distributed according to the event-type of the transaction. Events of any type may be published within the transaction; however TOPS will handle them as if they were events of the type for which the transaction was created. This guarantees that subscribers will receive all the transaction's events without knowing about their event type in advance. Otherwise, a participant in the transaction will not receive an event in the transactions when it hasn't subscribed to the event's type or content.

Published events are sent to the tx manager of all subscribers. Any tx manager that does not contain the txID will discard the event. A tx manager containing the txID also stores the scope of the transaction and thereby knows how to act accordingly. For instance, a subscriber with the txID marked as 'independent events' will not send any feedback to the event coordinator, and will not pass on 'abort' messages to the subscriber's application. A tx manager with a txID marked as a 'distributed transaction' will act as a participant in a distributed transaction and will communicate with the coordinator.

During execution of a transaction any participant may choose to abort the transaction. If the one aborting is a publisher, the abort is sent, as any other event is, to all tx coordinators. This will cause both distributed transaction subscribers and local transaction subscribers to abort. On the other hand, if a subscriber aborts, only its scope is aborted. In the case a local transaction subscriber aborts, only its own events are rolled back. In the case a distributed transaction subscriber aborts, events at all other distributed transaction subscriber are rolled back as well; however, local transaction subscribers are not rolled back.

Algorithm pseudo code

Algorithm for the publisher's logic:

```
BEGIN PublishTx(txID, event)
  Using the txID determine the location of the 'event coordinator'.
  IF the 'event coordinator' is here (publisher) THEN
    Send internal event to 'event coordinator'.
    Give event a sequence number.
    Using the txID determine the transaction's event type (evType).
    Publish event to all 'tx coordinators' including sequence number.
  ELSE    // 'event coordinator' is at rendezvous node
    Publish event to 'event coordinator' (at rendezvous node).
  END
END
```

Algorithm for the rendezvous node's logic:

```
BEGIN HandlePublishTx (txID, event)
  Using the txID determine the location of the 'event coordinator'.

  IF the 'event coordinator' is here (rendezvous node) THEN
    Send internal event to 'event coordinator'.
    Give event a sequence number.
    Using the txID determine the transaction's event type (evType).
  END

  IF 'tx coordinator' here contains txID THEN
    Send internal event to 'tx coordinator'.
    IF sequence number is wrong THEN abort transaction.
    Publish event to all 'sub-tx executors' including sequence number.
  END

  Publish event to all 'tx coordinators' including sequence number.
END
```

```
BEGIN HandleAbortTx (txID)
  Test constraints.
  IF constraints pass THEN
    Store participants.
    tx status = ok.
  END
END
```

Algorithm for the subscriber's logic:

```
BEGIN HandlePublishTx(txID, event)
  IF txID is unrecognized THEN RETURN

  Using the txID determine the scope.
  IF scope = 'get distributed' THEN
    Send internal event to 'sub-tx executor'.
    IF sequence number is wrong THEN send 'abort' to 'tx coord'.
    Have application process event.
    IF processing fails THEN send 'abort' to 'tx coord'
  ELSE IF scope = 'get local' THEN
    Send internal event to 'tx manager'.
    Send internal event to 'sub-tx executor'.
    IF sequence number is wrong THEN abort then transaction.
    Have application process event.
    IF processing fails THEN send abort then transaction.
  ELSE IF scope = 'get non-tx' THEN
    Send internal event to 'tx manager'.
    Send internal event to 'sub-tx executor'.
    Have application process event.
  END
END
```

4.6.3 Transaction Commitment

Commitment is initiated by instruction of a publisher. For local transactions, the 2PC protocol is not necessary. The tx coordinators of all participants receive a single commit event from the event coordinator and commit independently. For distributed transactions, the tx coordinator activates the 2PC protocol. At both phases of the 2PC protocol the constraints are tested and the transaction is aborted if they are not completely fulfilled. Testing of the constraints of the distributed transaction does not take non-participants into account, which may or may not commit the transaction locally. For a mixed transaction containing both local and distributed subscribers, the tx coordinator of the distributed transaction will initiate the 2PC protocol, and all local tx coordinators simply send a commit message. The outcome of this situation may be that all distributed subscribers abort while local subscribers commit successfully.

4.7 Application Programming Interface

TOPS exposes two interfaces, one for the p/s manager and one for the tx manager. Table 4-B lists the fundamental methods in each interface and their parameters (minor methods such as methods to enlist callbacks are omitted). For clarity, methods in the tx

interface have the 'Tx' suffix, and the 'On' prefix is used for callbacks that are called by the middleware and are to be implemented by the application.

Table 4-B: TOPS API

	P/S interface	TX interface
Publisher	Advertise(et) Publish(ev, et)	txID = BeginTx(et, txt, access, scope limit, constraints) PublishTx(txID, ev) CommitTx(txID)
Subscriber	Subscribe(et,filters) Unsubscribe(et) OnNotify(ev, et)	OnJoinTx(txID, et, scope limit) OnNotifyTx(txID, ev)
Common to publishers and subscribers		AbortTx (txID) OnAbortTx (txID) OnPrepareTx (txID) OnCommitTx (txID) OnCompensateTx (txID)

Parameter abbreviations:

et = event type ev = event

txt = transaction type txID = transaction ID

4.7.1 The Influence of TOPS Concepts on the API

The following bullets highlight the main points of interest regarding the API:

- Recall that there is no distinction between a standard pub/sub event-type and a transactional pub/sub event-type. For this reason, Advertise() and Subscribe() do not have equivalents in the tx interface.
- When creating a transaction using BeginTx(), the event-type is given (not an actual event). All events belonging to this transaction will be treated as of this type. A subscription to that event-type is automatically a subscription to transactions created with that event-type.
- All methods in the tx interface (except BeginTx()) receive the txID as parameter in order to indicate what transaction the call is referring to.
- PublishTx() doesn't require an event-type. However, since the event-type may still be of interest to the subscribers' applications, it may be included as data.

4.7.2 Use Case #1

The use of the API is demonstrated using the following simple scenario. Site S1 advertises event-type ET1 that site S2 subscribes to. Site S1 creates a single-publisher

distributed transaction of type ET1 and publishes two events: EV1 and EV2. Site S2 enlists in the transaction and receives the events. Site S1 requests to commit the transaction that is consequently committed successfully.

This scenario is carried out using the API as follows, and is depicted in Figure 4-O. First, S1 calls:

S1: Advertise(ET1)

to define the new event-type ET1. Site S2 now subscribes to the event-type ET1 by calling

S2: Subscribe(ET1, filters)

giving the event-type ET1 and content filters as parameters. Note that no special subscription is required for transactions, as they are supported inherently in all subscriptions. Until this point, the API calls were prerequisites of pub/sub and were not directly part of a transaction. To begin the transaction, S1 calls:

S1: BeginTx(ET1, distributed, private, limit none, none)

which returns a transaction id (TXID1). The parameters of BeginTx() were discussed in section 4.6. The middleware at S2 now calls:

Middleware: OnJoinTx(TXID1, ET1)

which returns the requested scope. Assuming the transaction's constraints given at BeginTx() are fulfilled, the middleware now internally creates the transaction. At this point S1 may publish events EV1 and EV2 by calling

S1: PublishTx(TXID1, EV1) and

S1: PublishTx(TXID1, EV2).

The event-type of these events is irrelevant to the middleware since any event published to the transaction will be regarded as type ET1. Still, the event-type is passed as data since it may be relevant to the receiving application. After each event is published, the middleware at S2 calls

Middleware: OnNotifyTx(TXID1, EV1) and

Middleware: OnNotifyTx(TXID1, EV2)

respectively. Site S1 has completed sending the transaction's events and now requests the transaction be committed by calling

S1: CommitTx(TXID1).

The middleware initiates a 2PC and invokes

Middleware: OnPrepareTx(TXID1)

at all subscribers, which in this simple case includes only S2. If all the participants prepare successfully, the middleware will invoke

Middleware: OnCommitTx(TXID1)

which concludes the transaction. Figure 4-O illustrates this process using a sequence diagram.

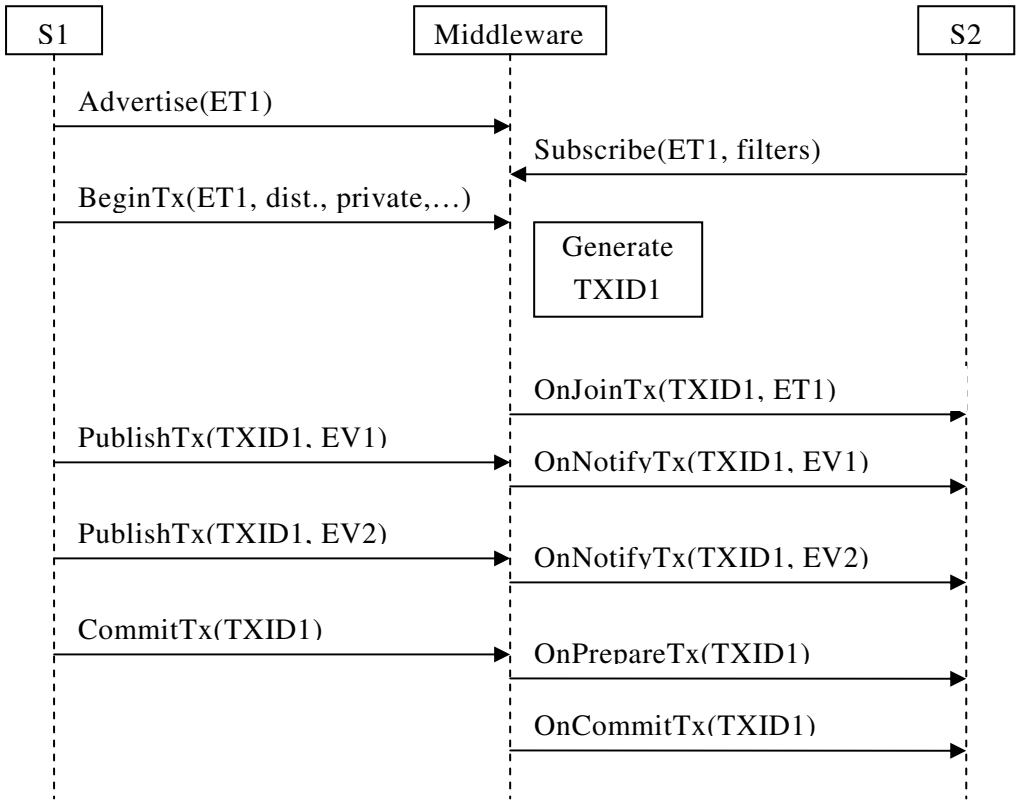


Figure 4-O: API Use Case #1 Sequence Diagram

4.7.3 Use Case #2

This use case is more complex and includes multiple publishers and subscribers and a subscriber (S3) with a non-transactional scope. Site S1 advertises event-type ET1 that sites S1, S2 and S3 subscribe to. Site S1 creates a multi-publisher distributed transaction of type ET1. The transaction is given a constraint requiring at least two participants. S1 (the transaction initiator) and S2 join the transaction whereas S3 does not join, yet it still requests to receive the events (which is allowed when the transaction is created with the scope limitation 'limit none'). S1 now publishes event EV1 which S1 and S2 receive within the scope of the transaction, and S3 receives as a regular event. Next, S2 publishes event EV2 to the transaction which is received by all sites as EV1. Site S1 requests to commit the transaction that is consequently committed successfully. Since S3 didn't join the transaction it does not take part in the commit phase.

This scenario is carried out using the API as follows and is depicted in Figure 4-P. First, S1 calls:

S1: Advertise(ET1)

to define the new event type ET1. Sites S1, S2 and S3 now subscribes to the event-type ET1 by calling

S1: Subscribe(ET1, filters)

S2: Subscribe(ET1, filters)

S3: Subscribe(ET1, filters)

giving the event type ET1 and content filters as parameters. To begin the distributed transaction, S1 calls:

S1: BeginTx(ET1, distributed, public, limit none, 'at least 2 participants')

which returns a transaction id TXID1. The 'public' scope parameter denotes a multi-publisher transaction. A scope limitation of 'limit none' is required to allow subscribers to receive the events outside the scope of the transaction (such as S3 in this scenario). The middleware at S1, S2 and S3 now calls:

S1 Middleware: OnJoinTx(TXID1, ET1) , return: 'as distributed' scope

S2 Middleware: OnJoinTx(TXID1, ET1) , return: 'as distributed' scope

S3 Middleware: OnJoinTx(TXID1, ET1) , return: 'as non-tx' scope

which returns the requested scope. S1 and S2 request the 'as distributed' scope and S3 requests the 'as non-transactional' scope. Since two sites (S1 and S2) are participating in the transaction, the transaction's constraints given at BeginTx() are fulfilled. The middleware now internally creates the transaction. At this point S1 publishes event EV1 by calling

S1: PublishTx(TXID1, EV1)

The middleware at S1 and S2 calls:

S1 Middleware: OnNotifyTx(TXID1, EV1)

S2 Middleware: OnNotifyTx(TXID1, EV1)

which is the transactional version of notification. The middleware at S3 calls:

S3 Middleware: OnNotify (EV1, ET1)

which is the standard version of notification. Now S2 publishes an event to the transaction. He is capable of doing so because he knows the transaction ID (TXID1). He uses it to publish event EV2 and calls:

S2: PublishTx(TXID1, EV2)

Similarly to EV1, the middleware at S1 and S2 calls:

S1 Middleware: OnNotifyTx(TXID1, EV2)

S2 Middleware: OnNotifyTx(TXID1, EV2)

and at S3:

S3 Middleware: OnNotify (EV2, ET1)

Note that the event-type is ET1! Even though EV2 is of type ET2, because EV2 was published to a transaction, it assumes the transaction event type which is ET1. Site S1 now requests the transaction be committed by calling

S1: CommitTx(TXID1).

The middleware initiates a 2PC and invokes

S1 Middleware: OnPrepareTx(TXID1)

S2 Middleware: OnPrepareTx(TXID1)

at all participating subscribers, which in this case includes S1 and S2 but not S3. Assuming all the participants prepare successfully, the middleware invokes

S1 Middleware: OnCommitTx(TXID1)

S2 Middleware: OnCommitTx(TXID1)

which concludes the transaction. Figure 4-P illustrates this process using a sequence diagram.

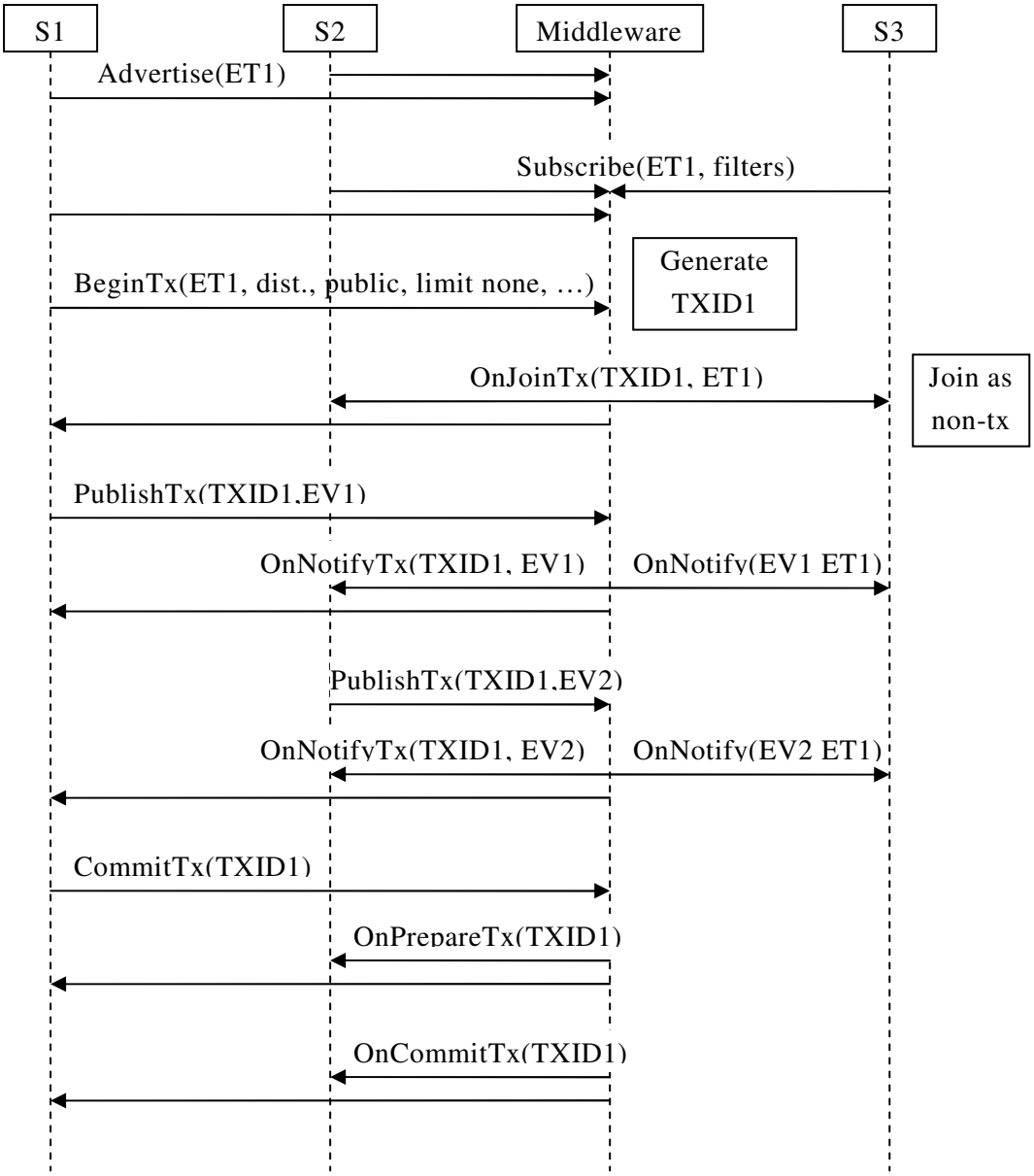


Figure 4-P: API Use Case #2 Sequence Diagram

5 Applying TOPS to Replication

Large distributed databases contain data on many servers each of which is capable of executing queries. It becomes increasingly difficult to synchronize between copies of data on different servers. One solution is to avoid the problem by storing data only once. The data is divided between the servers so that no data is stored on more than one server. In this case, reads and writes must be sent to the specific server that manages the data at hand. This solution has several weaknesses:

1. Not all servers are necessarily closely connected. Delays in communication will drastically degrade performance.
2. If many requests are placed for the same data, the server containing that data will become overloaded.
3. If a server fails, data stored on it is completely inaccessible since it is the only copy.

These drawbacks weaken the availability and the performance of the database system. These drawbacks may be solved using replication. Using replication, the same data is kept on several servers (replicas) any of which may be contacted in order to access the data. Referring back to the drawbacks stated above:

1. Since the data exists on several replicas, the closest one may be accessed thereby minimizing delays.
2. Load balancing of the replicas can moderate peaks and prevent any specific replica from becoming overloaded.
3. Data becomes inaccessible only if all replicas fail.

5.1 Methods of Replication

Several methods of replication exist [23]. The different methods of replication can be divided into two primary categories: *synchronous (eager)* methods and *asynchronous (lazy)* methods of replication. In synchronous replication, changes to data are synchronized between all replicas before the change is committed. This ensures that the change will either take effect at all replicas or will not take effect at all. Conversely, asynchronous replication applies the change at a single replica, and within a short period of time propagates the change to all other replicas. Of course, a set of changes applied to a single replica as a transaction, must be applied on all other replicas as a transaction as well.

5.1.1 Asynchronous Methods of Replication

Asynchronous replication does not involve coordination with remote sites while applying changes. Updating data is relatively quick since only one replica is updated. No remote resources need to be locked and the local update need not wait for communication with remote sites in order to commit a local transaction. Deadlocks between replicas, due to separate transactions locking each other's resources, are also avoided. On the down side, asynchronous replication does not provide atomicity or isolation. Due to the fact that updates are committed at the initiator and eventually propagated to all replicas, it is possible to find the same data on different replicas in a different state. Two methods of asynchronous replicas are *peer-to-peer (multi-master)* and *primary site (primary master).*

Peer-to-Peer

This method allows any replica to update its copy of a data object. After changes are committed, they will be sent to all other replicas in order to inform them of the changes. Using this method, performance of both reads and writes is high since any (presumably the closest) replica may execute a query containing writes. Isolation is weak because two sites are capable of updating and committing different changes to the same data object before receiving updates from the other. Firstly, peer-to-peer replication must cope with situations in which it is unknown which update should take precedence. In order to minimize such undesirable situations, updates are often given a time stamp in order to prevent overrunning current data with outdated data from delayed updates. TOPS provides global ordering for transactional events (in the event coordinator component). Since TOPS focuses on transactions, global ordering for standard pub/sub was not looked into and is beyond the scope of this work. Secondly, even once precedence is resolved, two simultaneous reads of the same data from different replicas will return unequal results when one replica has not yet applied the update and the other replica has.

Primary Site

Primary site replication designates a single replica for updating. If any other replica wishes to perform an update it must request the primary site to do so. The primary site will apply the change locally, and then asynchronously propagate it to all the secondary replicas. Reads may be executed at any of the replicas; however, only reads at the primary site are guaranteed to be current. The advantage of primary site replication over peer-to-peer replication is achieving global serialization of updates for any specific data object. Serialization solves the problematic situation demonstrated above; however, isolation is still not achieved. Since data objects may be managed by different sites, transactions involving data objects at several sites may conflict. Figure 5-A

demonstrates such a case. Transaction tx1 updates ‘a’ and ‘b’ at replicas 1 and 2. Replica 1 is the primary site of ‘a’ and therefore replicates it to replica 3. Replica 2 is the primary site of ‘b’ and replicates it to replica 3. At the same time, tx2 involving ‘a’ and ‘b’ executes at replica 3 catching ‘a’ and ‘b’ in an intermediate state violating isolation. The algorithms discussed in section 5.2.6 avoid this problem.

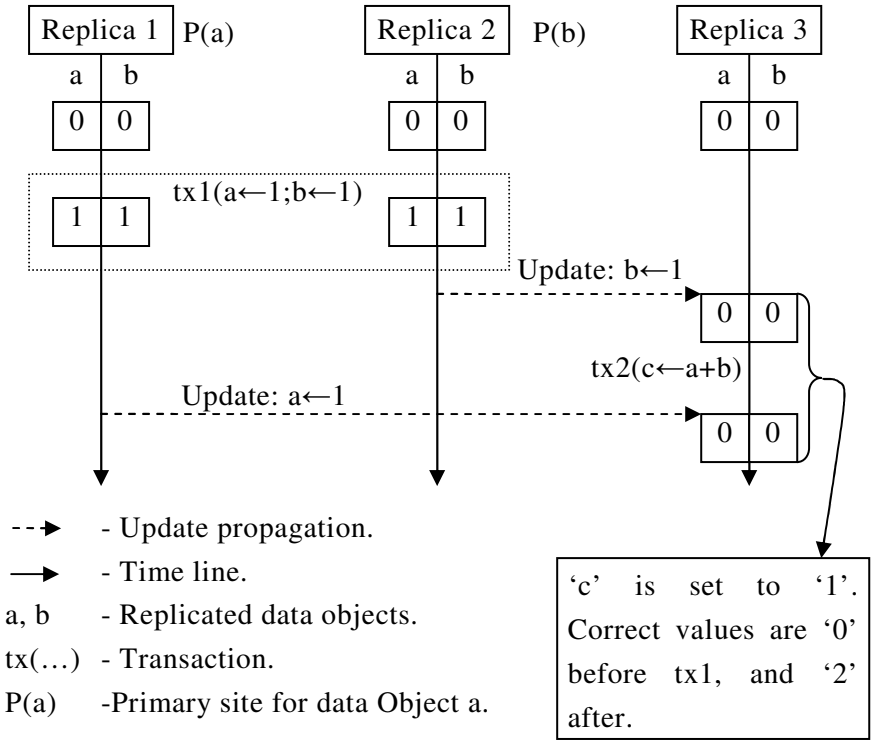


Figure 5-A: Primary Site Replication Isolation Violation

5.1.2 Synchronous Methods of Replication

Synchronous replication is achieved by performing updates within a distributed transaction. In order to ensure atomicity and isolation, the transaction must first lock relevant resources at all replicas performing the update, and then execute a commit protocol. Two methods of synchronous replication are *write-all read-any* and *quorums/voting*.

Write-All Read-Any

When updating data, the update is written to all replicas. Since all replicas are up-to-date, reads may be executed from any replica equivalently. Writes are costly since all replicas must be locked; however, reads are cheap since any replica may be queried. This method is suitable when the majority of queries only read data and a minority

require updating data, which is usually the case in ordinary applications. Upon failure of one of the replicas, updates will not be possible because not all replicas will have acknowledged the transaction. [20] gives an interesting perspective on the performance advantages of this method in relation to Quorums and other methods, arguing that it outperforms other replication methods in most all applications excluding applications with extreme update ratios (80%-90% of the operations are updates).

Quorums/Voting

By compromising the performance of reads, it is possible to simplify writes by updating only a subset of all replicas. Given n replicas, we may update only m of them ($1 \leq m < n$) and attach a version number to the data written. Since the reader requires the most current data, he must conduct a vote among at least $n - m + 1$ replicas in order to be certain that one of them is most up-to-date. The up-to-date replica is the one with the most advanced version number. Using voting, transactions performing updates will require locking of fewer resources and will most likely complete in less time. Nevertheless, the voting process causes reads to become significantly more expensive.

5.1.3 Replication Methods Conclusion

Several methods of replication exist each with its advantages and disadvantages. The basic tradeoff in all methods is between performance and data integrity. The most appropriate method for use is application specific, as well as handling the performance and correctness issues of the chosen replication method. Applications such as banking require absolute data integrity which is achieved by *global serialization*. A method of replication is considered to provide global serializability if the sequence of transactions executed at each replica yields the same outcome. Less demanding application compromise integrity, and use replication solutions that provide different levels of *relaxed serializability*. Such solutions commonly provide conflict resolution mechanisms.

Current databases solutions such as SQL-Server [14], Oracle [17] and DB2 [7] each provide their own replication solutions, some of which incorporate pub/sub concepts. Of the three, only Oracle supports synchronous replication. SQL-Server supports *snapshot replication* that creates a one-time copy of data, *merge replication* which is *asynchronous peer-to-peer*, and *transactional replication* which is *asynchronous primary site* (no global transaction context). DB2 supports asynchronous multi-master replication; however, not synchronous replication.

5.2 Implementing Replication Using TOPS

After presenting TOPS and reviewing several methods of replication, we will demonstrate how the different methods of replication may be implemented using TOPS.

Pub/sub, or at least the concept of pub/sub, is already commonly used for asynchronous replication. We take the use of pub/sub a step further by utilizing a transaction-oriented pub/sub middleware thereby supporting synchronous replication as well.

5.2.1 Asynchronous Peer-to-Peer

In peer-to-peer replication more than one replica (but perhaps not all) may update its local copy of data. All replicas are updated eventually, optimistically within a short period of time. First, replicas that are capable of updating their copy are defined as publishers and all replicas, including publishers, are defined as subscribers. This is necessary because updatable replicas must still receive updates from other updatable replicas. The transaction must be of multi-publisher access in order to support multiple publishers. Peer-to-peer replication may be implemented using either TOPS's local transactions or application managed transaction for which only pub/sub capabilities are utilized. The former is more robust and provides the application with the functionality of transactions at each replica. The latter may be used when the application prefers not to manage transaction within the middleware, in which case the standard pub/sub capabilities can be used to provide a means of data distribution. Both implementations support two variants:

- The initiating replica updates its own copy before notifying all other replicas. To achieve this, the initiating replica updates its copy independently of the middleware and then publishes the change with the appropriate event type in order to update all other replicas. When transactions are managed by the application, the application must first commit the transaction and only then approach the middleware for publication. When the transaction is managed by TOPS, this variant is achieved by creating a transaction of type 'local' with single publisher access. The scope limitation should be set to 'limit local' to enforce execution at subscribers as a transaction. A slight difference exists between transaction managed by the application and TOPS. When using TOPS the transaction is managed locally but events are sent out to replicas immediately rather than accumulating the events and sending them all at once when the initiator commits.
- The initiating replica publishes the update without first applying it locally; knowing that it will be updated once the publication reaches it as a subscriber. Updates at the initiating replica now require communicating with the rendezvous node which will slow the transaction execution at the initiator. Even though transaction management is centralized at the rendezvous node, the initiator remains independent of all other replicas. One advantage of this variant is that since all updates of the same data pass though the same rendezvous node, global serializations is achieved thereby moderating the problem of simultaneous updates initiated at separate replicas. A property of this variant worth noting is that the first replica to update may not be the initiating replica.

The initiator is regarded as any other subscriber and the order of reception is determined by the underlying pub/sub system.

Values of Transaction Properties

- Transaction type : Local.
- Access : Public.
- Scope limitation : Limit local.
- Constraints : None.

5.2.2 Asynchronous Primary Site

Asynchronous replication using a primary site is a sub-case of peer-to-peer replication except that data may be updated only by a single site. The difference in pub/sub terms is that there is only one publisher. Implementation using TOPS is the same as peer-to-peer with the slight adjustment of defining the transaction as single publisher. Internally, this will cause the tx manager to reside on the initiating replica instead of the rendezvous node, thereby preventing other publishers from contributing events to the transaction.

Values of Transaction Properties

- Transaction type : Local.
- Access : Private.
- Scope limitation : Limit local.
- Constraints : None.

5.2.3 Synchronous Read-Any Write-All

To obtain synchronous read-any write-all replication, all replicas must be updated together. To achieve this, distributed transactions are used. In order to assure that all replicas are updated, the transaction is created with a constraint requiring all replicas to participate. This may be obtained by predefining a list of all replicas' (encrypted) ids, under the not unreasonable assumption that any replica initiating updates must be aware of the other replicas' existence. This list is given to the transaction as a constraint and can be used to allow participation only of replicas on the list. Requiring participation of all replicas on the list is obtained by joining two constraints, the first being participants from the list and the second being a minimum amount of participants equal to the

number of ids in the list (defined as *m* in 5.1.2). Since constraints are checked at transaction commitment (as well as at creation), it is guaranteed that all replicas are updated synchronously.

The transaction's access may be single publisher or multi-publisher. Commonly, standard replication will require only single publisher access.

It is not necessary that all replicas be synchronous. It is possible to define a list of *m* synchronous replicas that are required to participate in the transaction, and still allow additional asynchronous replicas to receive the transaction at local scope. To support asynchronous replicas, the scope limitation is to be lowered to 'limit local' or 'limit none'. Note that the asynchronous replicas are not participant in the distributed transaction and, therefore, do not participate in constraint validation. For example, if a distributed transaction requires the participation of A, and A joins with 'local' scope, the transaction will fail creation because the constraints were not met. Furthermore, if a distributed transaction with a scope limitation of 'limit local' constrains participation to only A and B, C may join with 'get as local' scope and will not be rejected.

Values of Transaction Properties

- Transaction type : Distributed.
- Access : Private or public.
  Private access is commonly sufficient (single publisher).
- Scope limitation : Limit distributed.
- Constraints : All replicas must participate
  (validated at commitment).

5.2.4 Synchronous Quorums/Voting

Voting is quite similar to read-any write-all replication, the difference being that voting only requires *n* out of the *m* replica on the id list. The condition previously requiring *m* participants is simply changed to *n*. All other parameters remain the same as read-any write-all replication.

Values of Transaction Properties

- Transaction type : Distributed.
- Access : Private or public.

Private access is commonly sufficient (single publisher).

- Scope limitation : Limit distributed.
- Constraints : All least *n* replicas must participate.
  (validated at commitment).

5.2.5 Combined Synchronous and Asynchronous

In the above replication methods, replicas are either all synchronous or all asynchronous. Since TOPS allows each subscriber to determine its own scope, it is possible to combine both synchronous and asynchronous replications. Replicas joining the transaction with 'get as distributed' scope are synchronous replicas and replicas joining the transaction with 'get as local' scope are asynchronous replicas. Of course, the transaction initiator must specify a scope limitation of 'limit local' or 'limit none'. This flexibility is the first step towards being able to implement advanced replication methods that attempt to utilize the advantages of both asynchronous and asynchronous replication. This is further discussed in Section 5.3.

For instance, the transaction may require *m* particular replicas to participate synchronously; however other asynchronous replicas may exist which receive the data updates. In this scenario, an application interested in accessing a synchronous copy must approach one of the *m* synchronous sites and an application not requiring a synchronous copy may approach the closest replica which may be either synchronous or asynchronous.

Values of Transaction Properties

- Transaction type : Distributed.
- Access : Private or public.
  Private access is commonly sufficient (single publisher).
- Scope limitation : Limit local.
- Constraints : All least *m* replicas must participate.

5.2.6 Conclusion

We have shown how several different methods of replication can be implemented using TOPS. Note that we were required to make use of TOPS's unique features such as local transactions, access, and scope limitation. In the next section we give a theoretical analysis on implementation of some more advanced replication methods using pub/sub.

5.3 Advanced Replication Methods

The fundamental replication algorithms presented in section 5.2 are either synchronous, providing very strong data integrity with problematic performance, or asynchronous, providing very little data integrity with good performance. These fundamental algorithms were acceptable for applications up to approximately a decade ago. Since application scaling advanced rapidly, large-scale real life applications required a better compromise of integrity and performance. Much work has since been inverted in this subject and several new algorithms exist.

Most algorithms improve integrity and performance by sacrificing a third feature: the flexibility of the logical network layout used for propagation of data updates. The algorithms in section 5.2 have no restriction regarding the manner in which updates must be disseminated. By limiting update channels to 'legal' paths, it is possible to considerably improve data integrity (and even achieve global serialization) while the vast majority of updates are asynchronous.

The goal of this chapter is to provide a discussion regarding the applicability of transactional publish/subscribe to sophisticated replication algorithms. Two such algorithms are described in sections 5.3.1 and 5.3.2. To give a fair balance of the applicability to transactional publish/subscribe, we give one example (section 5.3.1) that is not suitable to be implemented using pub/sub, and another that is quite suitable for implementation over pub/sub (section 5.3.2). This section is not intended to be a comprehensive survey of replication algorithms; there are many other algorithms that are not mentioned here and their pub/sub implementation is a topic for future research.

5.3.1 The BackEdge Protocol

In [2] Y. Breitbart et.al. begin by presenting two primary site asynchronous replication protocols (DAG(WT) and DAG(T)) that guarantee serializability if replicated data flows between replicas in a directed acyclic graph (DAG). Next, they propose the BackEdge protocol which is an extension to the above two protocol. The BackEdge

protocol provides serializability even in graphs containing cycles. This is achieved by mixing in synchronous behavior in some particular cases, as will be further explained.

To begin, it is explained how the DAG refers to the topology of replication sites and data. A *copy-graph* is a directed graph representing the flow of data between replication sites. Each replication site is a node in the graph. Since these algorithms refer to primary site, any data item has only one primary copy. The copy graph contains a directed edge from site S_i to site S_j if S_i is the primary site of any data item that is copied (replicated) to site S_j. It is not necessary for communication from S_i to S_j to be direct. Figure 5-B shows an example copy graph. In this example, site S_1 is the master of data item 'a' that is replicated to S_3 and S_4, S_2 is the master of 'c' that is replicates to S_3, and S_3 is the master of data item 'b' that is replicated to S_4. If the copy graph contains no cycles (it is a DAG), the DAG(WT) and DAG(T) protocols may be applied.

Data item	Primary site	Replicas
a	S_1	S_3, S_4
b	S_3	S_4
c	S_2	S_3

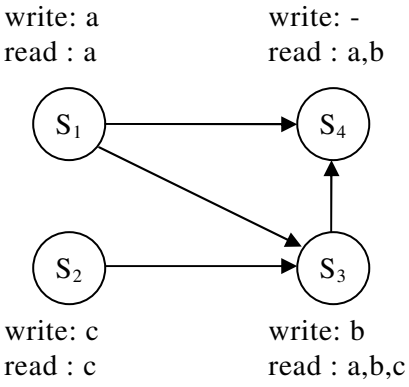


Figure 5-B: Example Replication Copy Graph

The DAG(WT) Protocol

The first protocol DAG(WT) (Directed Acyclic Graph Without Timestamps) achieves serializability by limiting the paths of communication between sites to a tree. Any two nodes in the copy graph with a parent-child relationship will have a parent-descendent relationship in the propagation tree. The purpose of the tree is to eliminate the possibility of a site having two parents. Having only one source of updates allows a site to maintain a serial execution. The downside of this protocol is that replicated data must travel through extra intermediate sites before reaching its final destination. A tree for the copy graph in Figure 5-B may look something like this:

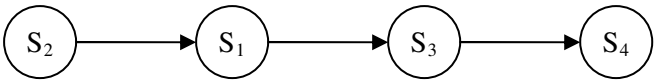


Figure 5-C: DAG(WT) Copy Tree

The DAG(T) Protocol

The second protocol DAG(T) (Directed Acyclic Graph with Timestamps) allows communication over any edge of the copy graph and uses timestamps to serialize updates among all sites. A description of the timestamp mechanism is not given here. It suffices to say that the downside of this algorithm is that the root of the tree must constantly send heartbeat messages and idle sites must send dummy updates in order to keep the timestamp running in order to insure progress.

The BackEdge Protocol

The two protocols above are limited to copy graphs that are a DAG. Suppose that it is necessary to add to Figure 5-B a data item 'd' at S_4 that is replicated to S_1 (Figure 5-D). The new edge added is a backedge because points in the backwards direction of the DAG. After removing all backedges from the graph we remain with a DAG.

Data Item	Primary site	Replicas
a	S_1	S_3, S_4
b	S_3	S_4
c	S_2	S_3
d	S_4	S_1

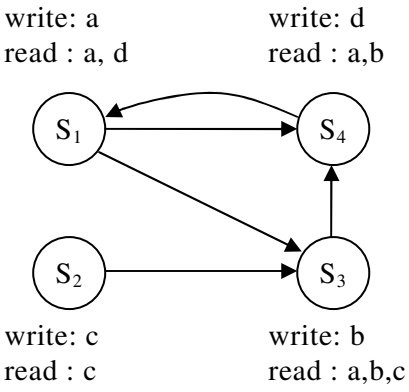


Figure 5-D: Example Replication Copy Graph with Back-Edge

Consider the situation in which transaction T_1 at S_1 reads 'd' and updates 'a'. At the same time, transaction T_2 at S_4 reads 'a' and updates 'd'. This execution is non-

serializable because S_1 executes the transaction in the order (T_1, T_2) and S_4 executes the transactions in the order (T_2, T_1).

The BackEdge protocol extends the DAG(WT) and DAG(T) protocols and offers a solution to such graphs that are not a DAG (they contain backedges). The principle of the protocol is as follows:

1. Identify the minimal set of back edges in the graph. Ignoring the backedges we are left with a DAG.
2. When replicating over non-backedge edges, use the DAG(WT) or DAG(T) asynchronous replication protocols as described above.
3. When replicating over a backedge, use synchronous replication for all of the site's parents. After the synchronous replication succeeds, use asynchronous replication for the site's descendants.

We now revisit the example above with transactions T_1 and T_2 this time using the BackEdge protocol. Since T_2 involves replication over a backedge, it will do so synchronously. S_1 updates 'a' asynchronously therefore it updates 'a' immediately (commits T_1) and then replicates it. S_4 updates 'd' synchronously therefore T_2 holds on to locks without being committed and replicates 'd' to S_1. The order of execution at S_1 is (T_1, T_2). S_1 holds on to locks as well and replicates the transaction down the tree. All the sites on the branch to S_4 hold locks and replicate. By the time T_2 makes its way back to S_4, T_1 would have reached S_4. In this example S_4 will find that T_2's lock interfere with T_1. T_2 is therefore aborted. If T_2 had made its way back to S_4 without being aborted, a distributed commit protocol (such as 2PC) would be used to commit T_2 at all involved sites. If the commit is successful, S_4 replicates the transaction to all its descendants (in our example there are none).

Applicability of the BackEdge Protocol to Pub/Sub

Before going into using pub/sub for the BackEdge protocol, we briefly discuss using pub/sub for the DAG(WT) and DAG(T) protocols. In both protocols the copy graph is a DAG, each parent site publishes an event-type for each data type it replicates, and each child site subscribes to these event-type. A restriction exists on the pub/sub implementation because the DAG(WT) and DAG(T) protocols rely on queues that guarantee ordering. A common asynchronous pub/sub implementation may not guarantee ordering of events. A pub/sub architecture that uses hierarchal routing (such as Hermes) can easily provide global ordering by having the rendezvous broker number messages belonging to each event-type. This still isn't equivalent to total order queues since total order queues guarantee ordering at the receiver. For pub/sub to provide this,

a component must be added at the subscriber that orders messages according to the rendezvous broker's numbering.

Now to the BackEdge protocol, a transaction traversing a backedge will be passed through all of the originating sites in a sequential order. Sequential/chain propagation of events is quite the opposite of the fundamental concept of pub/sub in which all subscribers essentially reside at the same level and are all reached in parallel via multicast trees. Maintaining the sequential order is mandatory in order to achieve serializability. If, for instance, a distributed transaction was used to update all parent sites, the transaction will arrive at each site's queue at a different location within the total order thereby violating serializability. Communication down the DAG may be accomplished by means of pub/sub, however, since the backedges may not be pub/sub, pub/sub does not provide a full solution.

Due to the reasons state above, we conclude that a pub/sub middleware is inappropriate for implementing the BackEdge protocol.

5.3.2 Replication with Serializability

In [8] a set of three replication protocols is introduced. Alongside the significant advantages of synchronous replication, its high performance cost is recognized. The protocols combine eager replication and lazy replication aspects in order to achieve the following advantages:

1. **High level of consistency**. Each of the three protocols provides a different level of consistency. The Replication with Serialization (SER) protocol is one copy serializable, which is the highest level of consistency and is equivalent to distributed transactions using two-phase-locking.
2. **Simplicity of implementation**. Previously suggested one-copy-serializable protocols were never implemented in commercial products due to their complexity.
3. **High performance**. The suggested protocols improve performance by minimizing the amount of messages needed to be sent, by minimizing the amount of time locks must be held, and by avoiding deadlocks. A most interesting aspect of the SER protocol from the perspective of transactional pub/sub is the fact that serialization is achieved without requiring an expensive 2PC.

The protocols utilize group communication primitives to obtain consistency. The group communication primitive used is a total order queuing service that provides:

1. Multicast queues.
2. Message delivery guarantee.
3. Total order guarantee.

All three replication protocols introduced are essentially relatively minor variations of the same thing. Here we will be focusing on the SER protocol which is the variant with the highest level of consistency.

The Replication with Serializability (SER) protocol

Since 'read' operations are generally much more common than 'write' operations, the protocol applies the Read One Write All Available (ROWAA) replication method. Using this method, 'read' operations are very cheap since no communication is required with any other sites. When performing a write operation, the update must be sent to all other replicas.

The first principle of the protocol is that each transaction is divided into two phases: the 'read' phase and the 'write' phase. All the 'read' operations of the transaction are performed up front and lock the relevant data items. Once completed, the protocol moves on to handling the 'write' operations. The set of a transaction's 'write' operations is named a *write-set*. This write-set is sent to a total-order queue (as discussed further on) in a single message to be applied atomically at all sites. This single message per transaction requires much less messages than some other protocols that require a message per 'write' operation.

The primary concept of the protocol is to have the replication manager at each site act in such a way that when the updates received at all sites are the same, the updates applied at all sites are also the same. Using a total order queue mentioned above, it is assured the updates will be received at all sites and in the same order. In order to maintain a global total order, the protocol prohibits local updates. The initiating site must publish its updates (write-sets) into the total order queue and only when they are received back from the queue he may apply his updates locally. Before the updates make it back to the initiator, there are most likely other updates in the queue from other sites that must be handled first. These updates may require the initiator to abort his transaction. Updates earlier in the queue take precedence over updates later in the queue. Once the updates are received back at the initiator and it was not aborted, the initiator sends a single 'commit' message to all sites. Note that the abort and commit message are not sent in the total order queue. Since:

1. All sites contain the same data, and
2. perform the same updates (and in the same order), and
3. Submit all updates to the total order queue, and
4. Discard any initiated updates that conflict with updates submitted earlier to the total order queue,

we can say firstly that the protocol is one-copy-serializable, and secondly that it is suffice to send each site a single ‘commit’ or ‘abort’ message instead of a full 2PC.

To simply summarize the protocol, consider a site not initiating any transactions. Such a site will perform the following simplified steps:

1. Withdraw the next write-set from the queue.
2. *Lock phase*- Lock the write-set’s resources atomically.
3. *Write phase*- Apply changes without committing.
4. *Termination phase*- Once an ‘abort’/‘commit’ message is received from the initiator, rollback/commit the changes and release all locks.
5. Return to step 1.

The logic at a site initiating transactions is slightly more complex. When a transaction is initiated, all ‘read’ locks are first obtained (*Read phase*). If the transaction contains only read operation then the transaction is committed. If the transaction also contains ‘writes’ operations the write set is published to the total-order-queue (*Send phase*). Handling updates from the queue is now performed as follows:

1. Withdraw the next write-set from the queue.
2. *Lock phase*- Lock the write-set’s resources atomically.
    2.1. Grant locks:
        2.1.1. If data items are not locked, grant locks.
        2.1.2. If a lock exists and belongs to a transaction that is waiting for a ‘abort’/‘commit’ message, then postpone the lock until the message arrives.
        2.1.3. If a lock exists and belongs to a locally initiated transaction for which the write set has not arrived, then abort the transaction. If the write-set of the aborted transaction has been transmitted then send an ‘abort’ message to all sites.
    2.2. If all locks are granted then send a ‘commit’ message to all sites.
3. *Write phase*- Apply changes without committing.
4. *Termination phase*- Once an ‘abort’/‘commit’ message is received from the initiator, rollback/commit the changes and release all locks.
5. Return to step 1.

The main disadvantage of the SER protocol is that transaction with mixed ‘read’ and ‘write’ operations hold on to read lock for a long time – until the write-set arrives back from the total-order-queue. Additionally, since local transactions are always aborted

upon conflict, there may be situation in which a local transaction is continuously aborted.

Applicability of the SER Protocol to Pub/Sub

As discussed in the BackEdge protocol, the SER protocol requires total order delivery. Even though not naturally supported, a pub/sub architecture that uses hierarchal routing may be upgraded to provide total order delivery within an event-type. This is not an actual restriction, because SER uses a single total-order queue which is equivalent to using a single event-type. Ordering within a transaction is already supported (see section 4.5.3) ordering between transactions is not. Functionality must be added to the transaction manager to allow only one transaction at any given time for this event-type.

In a pub/sub implementation, pub/sub will replace the total order queue and the communication required for sending 'commits' and 'aborts'. Each site subscribes to its own transactions and will publish its write-sets into a pub/sub transaction. All other replication sites are subscribers as well. After the publisher receives the events back from the middleware and processes them, he will abort or commit the transaction. The advantage of pub/sub here is that it frees the replication site of the need to know to whom 'commit'/'abort' messages must be sent.

In the SER protocol, the initiator receives and handles its own changes by receiving them from the queue, just as any other site does. In order to be implemented using pub/sub, the pub/sub solution must allow the publisher to subscribe to its own events and must allow the publisher to publish events without applying them locally (see TOPS's features of initiator participation and multiple publishers, 4.4.1).

Due to the above discussing, we conclude that implementing the SER algorithm over pub/sub is manageable. It is important to note that the SER protocol achieves one-copy-serializability *without* distributed transactions; therefore, it does not require a transactional pub/sub middleware. Essentially, this is the goal and the primary advantage of the SER algorithm. Transactional pub/sub is a generic infrastructure and must rely on distributed transactions and 2PL, SER's problem domain is specific to replication that is a more specific case. The SER algorithm assumes that:

1. At the point the protocol begins, all replicas are identical in the aspect of the data they contain.
2. All replicas receive the same exact changes.

These assumptions are correct for replication and therefore allow for a simplified algorithm in this specific case.

5.4 Summary

In the above section we presented how TOPS, by supporting both local and distributed transaction, can be used to implement several methods of replication. Two synchronous methods and two asynchronous methods were discussed. In order to benefit from the performance of asynchronous replication and still achieve serializability, more advanced methods (such as [2]) combine elements of both synchronous and asynchronous dissemination. For such methods, a middleware offering support for both synchronous and asynchronous replication is likely to be even more beneficial than it is for the fundamental replication methods we presented above.

Additionally we provide an early work discussion of more advanced replication algorithms that reach out towards providing global serialization (mostly) without requiring distributed transactions and the costly 2PC. We show how some algorithms are foreign to pub/sub, whereas, other are definitely candidates for being implemented using a pub/sub middleware such as TOPS. We also discussed the option of using transactional features of pub/sub for implementing replication and noted that replication algorithms tied specifically to the replication problem domain are problematic to implement using generic pub/sub transactions. Further research on this subject is necessary before reaching final conclusions.

6 Conclusion

Much work has been done on the subject of pub/sub middleware and distributed transactions, yet a joint middleware is relatively a new subject of study.

In this thesis we introduced TOPS that in our opinion takes the integration of transactions and pub/sub a step further. Our main goal was to introduce a customizable middleware that provides a large variety of features for both pub/sub and transactions, separately and primarily jointly. These features include: content-base pub/sub, distributed, local and mixed transactions, multi-publisher transactions, compensatable clients and more. Additionally, we further integrated transactions into pub/sub while minimizing the level of awareness required from the publishers and subscribers. We discussed: the benefits of these new features, design considerations, and an architecture to support the proposed new features.

Since our goal was to advance the support for transactions in pub/sub middleware, we found it appropriate to base our work on the design of an existing middleware. We found HTS to be the currently most suitable middleware for fulfilling our objectives, and therefore selected it to be the basis for our work. Even thought we refer to the Hermes pub/sub middleware used in HTS, TOPS (as well as HTS) are essentially capable of being integrated into any pub/sub middleware that used rendezvous or hierarchical routing (such as Siena).

We concluded by demonstrating the strengths of TOPS by presenting how it may be used to implement different methods of replication.

6.1 Future Work

In this section we briefly list some possibilities for future work on the subject of distributed transactions:

- **Performance Analysis**: TOPS as well as all of the transactional pub/sub designs presented in the related work do not give a performance analysis of pub/sub transactions. To take the work on the subject passed the stage of proof of concept, TOPS must be implemented, and the performance of its transactions must be compared to that of a non pub/sub distributed transaction system.
- **Security**: In [1] security issues in pub/sub are discussed for the first time. Pub/sub is introduced to *Role Based Access Control* (RBAC). This paper does

not discuss issues related to transactions in pub/sub which is currently open for research.

- **In Depth Study of Replication Protocols**: In section 5.2.6 we gave an overview of advanced replication protocols, giving only few examples. To fully investigate the applicability of transactional pub/sub to replication, a comprehensive review is required
- **Nested pub/sub transactions:** Nested transactions are transactions that are executed within the context of an existing transaction. Support for nested pub/sub transactions was not part of TOPS's design. Such a situation can occur, for instance, when a participant of an encompassing transaction creates a nested transaction in response to an event published within the encompassing transaction. Further study is necessary in order to add support for nested pub/sub transactions

7 References

[1] Bacon, J., Eyers, D. M., Singh, J., and Pietzuch, P. R. 2008. Access control in publish/subscribe systems. In *Proceedings of the Second international Conference on Distributed Event-Based Systems* (Rome, Italy, July 01 - 04, 2008). DEBS '08, vol. 332. ACM, New York, NY, 23-34.

[2] Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., and Silberschatz, A. 1999. Update propagation protocols for replicated databates. In *Proceedings of the 1999 ACM SIGMOD international Conference on Management of Data* (Philadelphia, Pennsylvania, United States, May 31 - June 03, 1999). SIGMOD '99. ACM, New York, NY, 97-108.

[3] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.* 19, 3 (Aug. 2001), 332-383.

[4] Domaschka, J., Reiser, H. P., and Hauck, F. J. 2007. Towards generic and middleware-independent support for replicated, distributed objects. In Proceedings of the 1st Workshop on Middleware-Application interaction: in Conjunction with Euro-Sys 2007 (Lisbon, Portugal, March 20 - 20, 2007). MAI '07, vol. 224. ACM Press, New York, NY, 43-48.

[5] Eugster, P. 2007. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.* 29, 1 (Jan. 2007), 6.

[6] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A. 2003. The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 2 (Jun. 2003), 114-131.

[7] Gu, L., Budd, L., Cayci, A., Hendricks, C., Purnell, M., and Rigdon, C. 2002. A Practical Guide to DB2 UDB Data Replication V8. IBM RedBook.

[8] Kemme, B. and Alonso, G. 2000. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.* 25, 3 (Sep. 2000), 333-379.

[9] Liebig, C. and Tai, S. 2001. Middleware Mediated Transactions. In *Proceedings of the Third international Symposium on Distributed Objects and Applications* (September 17 - 20, 2001). DOA. IEEE Computer Society, Washington, DC, 340.

[10] Liebig, C., Malva, M., and Buchmann, A. P. 2001. Integrating Notifications and Transactions: Concepts and X2TS Prototype. In *Revised Papers From the Second international Workshop on Engineering Distributed Objects* (November 02 - 03, 2000). W. Emmerich and S. Tai, Eds. Lecture Notes In Computer Science, vol. 1999. Springer-Verlag, London, 194-214.

[11] Muhl, G., Feige, L., and Pietzuch, P. R., 2006. *Distributed Event-Based Systems*. Springer, Verlang Berlin Heidelberg.

[12] Kifer, M., Brenstein, A., and Lewis P. M., "Implementing Dstributed Transactions", in *Database Systems: An Application Oriented Approach*, 2nd Edition, Addison-Wesley, 2005, pp. 1005-1038.

[13] Michlmayr, A. and Fenham, P. 2005. Integrating Distributed Object Transactions with Wide-Area Content-Based Publish/Subscribe Systems. In Proceedings of the Fourth international Workshop on Distributed Event-Based Systems (Debs) (Icdcsw'05) - Volume 04 (June 06 - 10, 2005). ICDCSW. IEEE Computer Society, Washington, DC, 398-403.

[14] Microsoft Corp.: Types of Replication Overview: *Microsoft Developer Network - SQL Server Developer Center, SQL Server 2005 Books Online* http://msdn2.microsoft.com/en-us/library/ms152531.aspx.

[15] Milan-Franco, J. M., Jiménez-Peris, R., Patiño-Martínez, M., and Kemme, B. 2004. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX international Conference on Middleware* (Toronto, Canada, October 18 - 22, 2004). Middleware Conference, vol. 78. Springer-Verlag New York, New York, NY, 175-194.

[16] Moro, G. and Viroli, M. 2001. Enabling Business Cooperation using a Publish-Subscribe Architecture Aware of Transactions. In *Proceedings of the 34th Annual Hawaii international Conference on System Sciences (Hicss-34)-Volume 9 - Volume 9* (January 03 - 06, 2001). HICSS. IEEE Computer Society, Washington, DC, 9086.

[17] Oracle Corp. Oracle Database Advanced Replication, 11g Release 1 (2007) http://download.oracle.com/docs/cd/B28359_01/server.111/b28326.pdf

[18] Patiño-Martinez, M., Jiménez-Peris, R., Kemme, B., and Alonso, G. 2005. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.* 23, 4 (Nov. 2005), 375-423.

[19] Pietzuch, P. R. and Bacon, J. 2002. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd international Conference on Distributed Computing Systems* (July 02 - 05, 2002). ICDCSW. IEEE Computer Society, Washington, DC, 611-618.

[20] Jiménez-Peris, R., Patiño-Martínez, M., Alonso G., and Kemme, B. 2003, Are quorums an alternative for data replication?, in *ACM Transactions on Database Systems (TODS),* v.28 n.3, p.257-294.

[21] Ramakrishnan, R., "Parallel and Distributed Databases", in *Database Management Systems*, International Edition, McGraw-Hill, 1999, pp. 750-763.

[22] Shatsky, Y., Gudes, E., and Gudes, E. 2008. TOPS: A new design for transactions in publish/subscribe middleware. In *Proceedings of the Second*

international Conference on Distributed Event-Based Systems (Rome, Italy, July 01 - 04, 2008). DEBS '08, vol. 332. ACM, New York, NY, 201-210.

[23] Son, S. H. 1988. Replicated data management in distributed database systems. SIGMOD Rec. 17, 4 (Nov. 1988), 62-69.

[24] Tai, S. and Rouvellou, I. 2000. Strategies for integrating messaging and distributed object transactions. In *IFIP/ACM international Conference on Distributed Systems Platforms* (New York, New York, United States, April 03 - 07, 2000). Middleware Conference. Springer-Verlag New York, Secaucus, NJ, 308-330.

[25] Tai, S., Mikalsen, T. A., Rouvellou, I., and Jr., S. M. 2001. Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages. In *Proceedings of the 5th IEEE international Conference on Enterprise Distributed Object Computing* (September 04 - 07, 2001). EDOC. IEEE Computer Society, Washington, DC, 105.

[26] Vargas, L., Pesonen, L. I., Gudes, E., and Bacon, J. 2007. Transactions in Content-Based Publish/Subscribe Middleware. In *Proceedings of the 27th international Conference on Distributed Computing Systems Workshops* (June 22 - 29, 2007). ICDCSW. IEEE Computer Society, Washington, DC, 68.

[27] Liu, X., Niv, G., Shenoy, P., Ramakrishnan, K. K., and Van der Merwe, "The Case for Semantic Aware Remote Replication", in *Proceedings of the second ACM workshop on Storage security and survivability*, p. 79-84, 2006.

[28] X/Open. 2007. "Distributed Transaction Processing: Reference Modal". Version 3.

8 Appendix A – HTS Application Simulator

This appendix contains more detailed information regarding the HTS application simulator developed, in addition to chapter 3. Chapter 3 discussed requirements and design, here the topic of implementation and interface capabilities are discussed.

8.1 Implementation of Simulator

8.1.1 Development Environment

The simulator application uses the same development environment as HTS because it runs directly over it. HTS contains no GUI therefore the widget toolkit, as listed in Table 8-A, was a choice made specifically for the simulator.

Table 8-A: Development Environment Specification

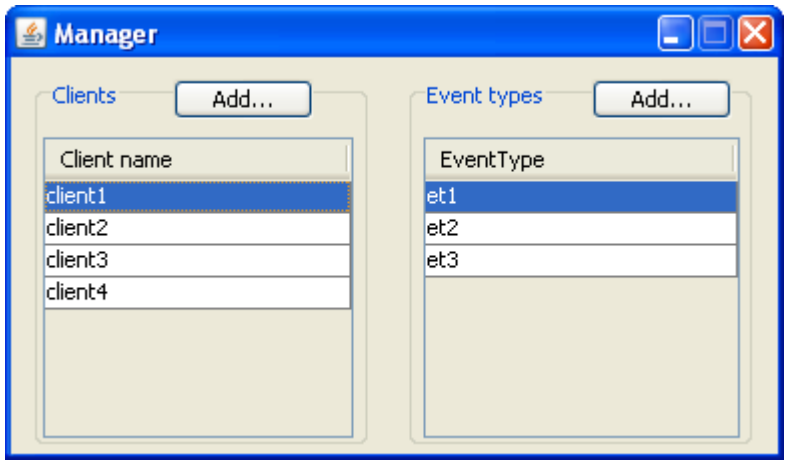
Tool type	Tool name	Version
Language	Java	JRE 6
Development environment	Eclipse	3.2.2
Widget toolkit	Swing	

8.2 User Interface Design

This section describes the visual specification of the windows the GUI layer contains in order to fulfill its requirements. The specification includes windows, controls and required behavior.

8.2.1 Main Window

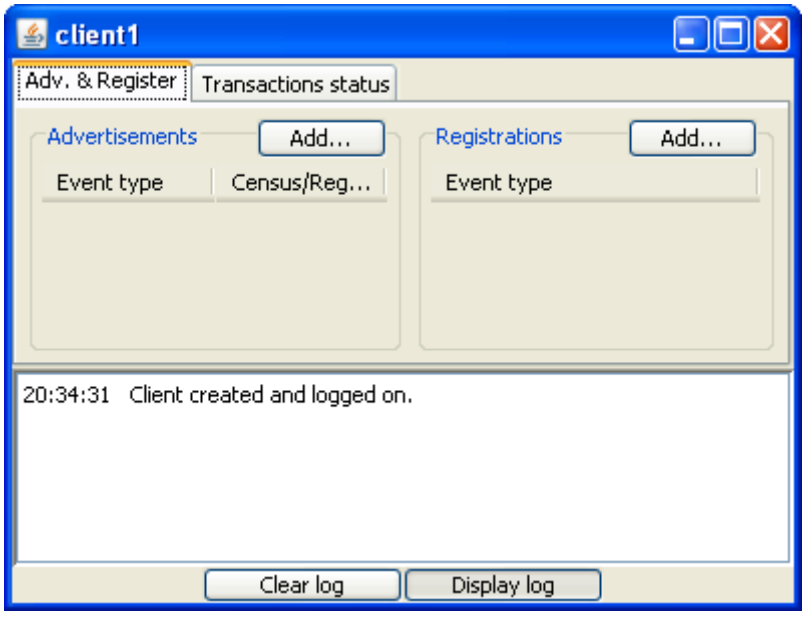
The main window is in charge of setting up the pub/sub environment, including creating clients (publishers and subscribers) and event-types. This window is managed by the *MainWindow* class. In the following example four clients were defined (client1, client2, client3 and client4) and three event-types were defined (et1, et2, et3).



Interface action/item	Description
"Add…" (Clients)	Create a new client.
"Add…" (Event types)	Create a new event type.
Double click client in list	Open client window.

8.2.2 Client Window- General

The client window manages the commands a publisher or subscriber may give: advertising, registering, publishing, subscribing and more. The window is split in to two tabs: the "advertise and register" tab, and the "transaction status" tab. These tabs are described in sections 8.2.3 and 8.2.4 respectively. The client window is managed by the *ClientWindow* class.



Interface action/item	Description
Title bar text	Shows the name of the client this window relates to (client1 in the above example).
Close window (X)	Hides the client window. The client remains active. The window can be reopened from the main window.
“Clear log”	Erase contents of log.
“Display log”	Turn display of log window on/off

As shown in the window, each client window contains a textual log. The log is viewable from both tabs.

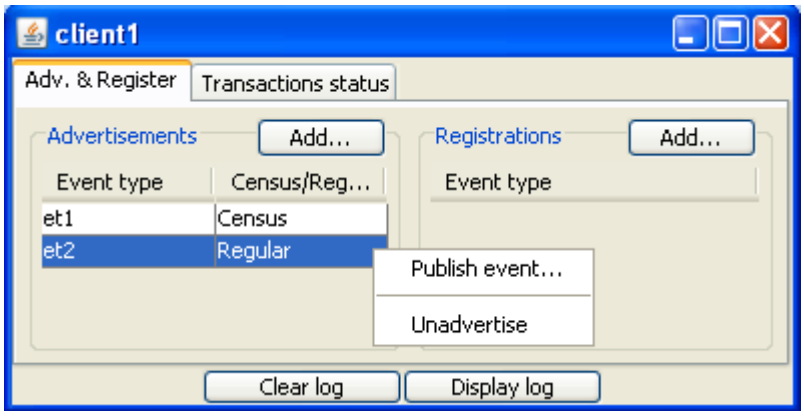
8.2.3 Client Window- Advertise and Register Tab

The “advertise and register” tab (“Adv. & Register”) provides the necessary GUI for a publisher to advertise an event-type and for a subscriber to register to an event-type.

Advertisement

The advertisements group is located in the left half of the “Adv.&Register” tab. The table lists all event-types this client advertised. The “Census/Reg…” column shows if the event was advertised as a census event or as a regular event. An event advertised as a census event is an event that initiates a transaction. When publishing a census event, a new transaction is created and its census phase is started. A regular event is an event that can be sent either outside a transaction as a non-transactional event, or within an already established transaction as part of the transaction execution.

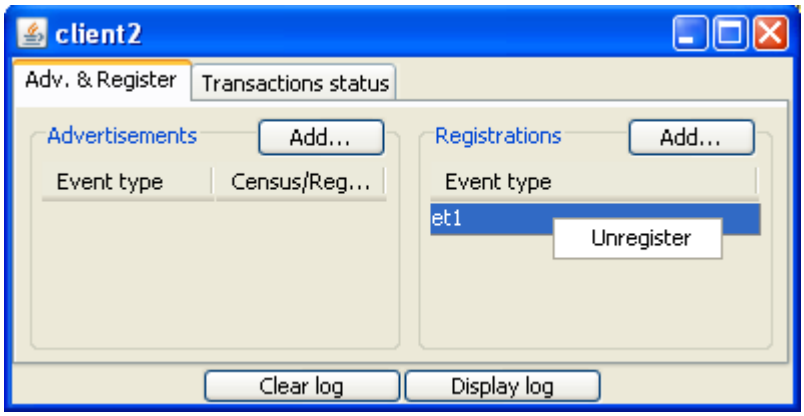
Pressing the “Add…” button opens the “New advertisement” dialog box in which an event-type is chosen (from the ones defined by the manager) and the advertisement type in chosen (census/regular). In the following example “client1” advertised “et1” as a census event and “et2” as a regular event. Right clicking a row (advertisement) opens a menu of actions which may be taken.



Interface action/item	Description
"Add…" (Advertisements)	Create a new census/regular advertisement.
"Publish event…"	Publish the selected event. If the event is a census event a transaction is started.
"Unadvertise"	Cancels the advertisement of the selected event and removes the event from this list.

Registration

The registrations group is located in right half of the "Adv.&register" tab. The table lists all events this client is registered to. Pressing the "Add…" button opens the new registration dialog box from which a registration can be created. A registration includes exactly one census event and any number of regular events. In the following example client "client2" creates a registration to events advertise by "client1". A registration may contain many regular event-types, however, only the census event-type is shown.

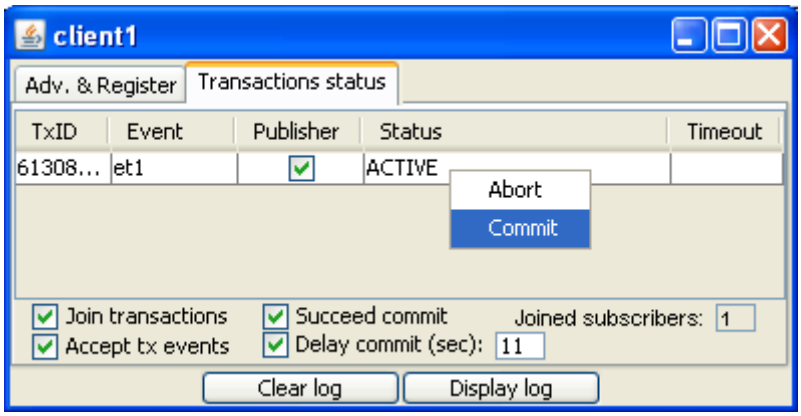


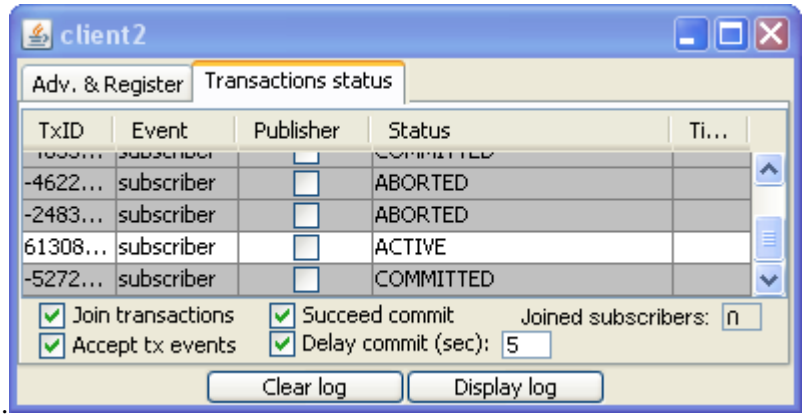
Interface action/item	Description
"Add…" (Advertisements)	Create a new registration.

"Unregister"	Cancels the selected registration and removes it from the list.

8.2.4 Client Window- Transaction Status Tab

The transaction status tab displays the status (census/active/committed etc.) of all transactions for the client, allows performing actions on a transaction (abort/commit), and provides the means to manipulate transaction behavior. The main part of this window is the transaction table. The transaction table lists transactions (one shown) in rows, and properties of the transactions in columns. The lower section of the window contains checkboxes which allow the user to change the way the client reacts to transactions. The window is common to all clients, however, some are relevant only to publishers and others are relevant only to subscribers. A more detailed description follows below. In the following example, "client1" published the census event "et1". "client2" joined the transaction and the transaction was created. "client2" has one active transaction and a history of previously committed and aborted transactions.





Interface action/item	Description
“TxID” (column)	The unique ID of the transaction. For a specific transaction, this number is identical at the publisher and at all subscribers.
“Event” (column)	At the publisher, this field shows the event-type of the census event. At the subscribers “subscriber” will appear in the field. Relevant only at publisher.
“Publisher” (column)	Checkbox is check if this client if the publisher of the transaction.
“Status” (column)	Displays transaction status which may be: CENSUS, CENSUS_UNFULFILLED, ACTIVE, COMMITING, COMMITED, ABORTED
“Timeout” (column)	Displays a countdown in the following cases: - Time left until the census phase times out. - Time left for subscribers to answer a commit request. Relevant only at publisher.
“Abort” (popup menu)	Instruct to abort the selected transaction.
“Commit” (popup menu)	Instruct to commit the selected transaction.
Join transactions (checkbox)	Determine if client will join transactions it is subscriber to. Relevant only at subscriber.
Accept tx events (checkbox)	Determine if client successfully handles events within transactions. An event not handled successfully causes an abort. Relevant only at subscriber.
Succeed commit (checkbox)	Determine if client successfully commits transactions. An unsuccessfully commit causes an abort.
Delay commit (checkbox)	Determine how long to delay this clients response to a commit request. If delayed longer than the publisher’s timeout, the transaction will abort. Relevant only at subscriber.
Joined subscribers (textbox)	Indicates how many subscribers joined the transaction. Relevant only at publisher.