A CHANGE NOTIFICATION FACILITY FOR WORKFLOW CLIENTS

By:

Christopher Waino Hockert

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

May 22, 2000

June 2000

Author_____
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by_____
Frans Kaashoek
Thesis Supervisor

Certified by_____
Sunil Sarin
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A CHANGE NOTIFICATION FACILITY FOR WORKFLOW CLIENTS

By:

Christopher Waino Hockert

Submitted to the Department of Electrical Engineering and Computer Science

May 22, 2000

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis describes the design and implementation of a change notification facility for InConcert, a workflow management system. Related systems that offer change information to clients are discussed, along with the design requirements for the proposed facility. The change notification facility uses TIB/Rendezvous, a publish/subscribe multicast architecture. Implementation discussion focuses on the integration of the change notification facility into InConcert's C++ based server and providing a Java client package to access the facility. The design and implementation of ActiveTaskList is discussed, a workflow client that uses the new facility. Finally, an analysis is performed on the facility's impact on the server, network, and clients that use it. This facility offers the desirable feature of having clients more quickly reflect the information on the server, and its performance impact was found to be moderate and manageable under levels of server activity that have been observed in practice.

Thesis Supervisor: Frans Kaashoek
Title: Associate Professor, EECS

Thesis Supervisor: Sunil Sarin
Title: Principal Architect, TIBCO Software Incorporated

Acknowledgements:

First I want to thank Sunil Sarin and Frans Kaashoek, my advisors. Sunil has been a great help answering my questions about InConcert, and has helped make this thesis possible. Professor Kaashoek has been a great help with shaping this document and having me answer some interesting questions, by which I have learned a great deal. Both Sunil and Frans took time out of their busy schedules and gave me feedback on this document, and I am grateful for their guidance.

Thanks to the people at InConcert, now TIBCO Software Incorporated: Alex Layton, Ian Jackson, Ed Black, Jon Gilman, Alireza Farhoush, Steve LeBlanc, Mike Register, Stephan Foerster, Patricia Lyga, Dick Vacca, Mike Keegan, Matt Hardesty, and John Bedell. These folks had to deal with all of my stupid questions, as well as some of my good ones. Additional thanks to all of the employees of InConcert past: Keith Gregory, Doug Knowles, Jenifer Tidwell, and Meryl Hlynka.

Thanks to all of my friends at MIT: Ankur Chandra, Laura Kwinn, Shawn Hwang, Jade Wang, Kavita Babal, and Chris Allen. They all have helped keep me sane. Special thanks to my good friend and roommate, Keith Santarelli, for tolerating my up-to-the-minute thesis progress updates. Yes Keith, it's done now, let's go drink!

Thanks to Paula Mickevich and Maria Sensale, my mothers in parallel universes. Thank you for keeping me employed during school and listening to my problems. You have been my good friends all the time I have been at MIT. Thank you for taking care of me.

Special thanks to Jeff, the best brother anyone could have. Thank you for being there and listening to me whine. Jeff will be getting married soon to a terrific woman, Katie, a very welcome addition to the Hockert family. Congratulations to both of them.

Special thanks to Mom and Dad, for their careful guidance throughout my life. Thank you for getting me here to MIT and supporting me with your prayers and Sunday noontime phone calls. I'm sorry I had to go to school on the other side of the country, but you have always been with me. Special thanks to Grandma, to whom I promise to paint the cupboards next time I am over. I love you dearly and should call you more often.

Extra special thanks to my beloved, Jennifer. She keeps going where others would have long ago given up. She was an inspiration to me to keep going with this thesis. Thank you for always being there for me. I'm sorry I had to spend an extra year here, but I'll be coming to New York soon!

That's all! Enough dilly-dally! On to the thesis!

Christopher Waino Hockert

Table of Contents:

Chapter 1: Introduction

The client/server model of distributed computing gives the appearance that the computer that you are sitting at is more powerful than it actually is. The ability to connect to services on a network for information makes the computer you are sitting in front of all the more powerful as a tool. One problem with many client/server systems is the fact that clients are given no notification that the information on the server has changed. Once the information leaves the server and reaches the client to be put in local memory, a copy has been made, and the information runs the risk of being out of date. It would be ideal if the server would let the clients know when its information has changed, and even more ideal if the server would notify the clients without each client having to check periodically.

This thesis focuses on the creation of a facility that provides clients with notification when information changes on the server. The facility is integrated into InConcert, a client/server based workflow management system developed by TIBCO Software Incorporated. The facility uses a publish/subscribe messaging architecture to distribute the messages to clients. An example application that uses the facility is described, along with other ideas on how this facility can create a more real-time, event-driven workflow environment.

The addition of the change notification facility to InConcert will allow clients, as well as other systems to act on the changing information in the InConcert workflow management system. It also improves the way applications that InConcert are developed, incorporating this information in order to make the applications better reflect the information on the server. We will see how this change notification facility will be more efficient than polling the server for changed information.

This thesis will review some related computer systems that offer change information to clients, and discuss the major issues with designing the change notification facility for InConcert. The server and client extensions are described in detail, along with their placement within the InConcert workflow management system. The design and implementation of ActiveTaskList is discussed, an application that uses the new change notification facility. An analysis is presented of how the change notification facility

affects the performance of the InConcert server, network and clients that use the facility. Finally, advice on how to develop clients that use the change notification facility is presented, along with possible design improvements to the facility.

Chapter 2: Related Work

This thesis focuses on bringing a change notification facility to a workflow management system, in order to find out how the technology will benefit from the ability to keep its clients more aware of change. The idea of a change notification facility for computer systems is not a new technology. This chapter will describe some systems that provide information on change, and allow clients access to that information. We will also look at how this information is made available to clients. This is not a complete list, we offer three representative examples.

2.1 Unreal Multi-User Network Game Protocol

Computer based entertainment systems have established the need to share information among networked computers. Popular first-person shooter games now allow users to connect over a network and play against each other in the same virtual space. The players can hunt for computer-generated enemies, or each other as they move through the virtual space.

The personal computer game Unreal, produced by Epic Games, Inc. uses an advanced, third-generation network game architecture to allow multiple users to play in a virtual space. At the center of the architecture is a server that maintains the complete game state. Clients that are involved in the game keep only part of the game state that is important to it, and send their movements to the server in the form of requests. Time passes in the virtual space in the form of variable length clock ticks. During a clock tick, the server processes the requests for movement on the clients, and sends each client updates of variables that directly affect the client. Clients use the update messages to update the local partial game state, and eventually refresh the display for the user playing the game. The server can make the optimization of only sending update messages for state that affects the client by analyzing the position of each client within the virtual space. It will only send update messages for an object's state if it is in the player's visual range [12].

The architecture of the messaging system is well designed and complex. It allows the ability for outside developers to create their own characters and scenes, along with

specifying how clients receive information on these new creations. It offers the ability to specify what data can be sent to clients reliably or unreliably. This is helpful for sending updates unreliably for state that is not vital to game play, like background decorations. Here the information could be lost with no effect on game play. In the cases where the update is vital, like when an enemy moves directly in front of you, the information can be sent reliably, so it will be sure to reach the client [12].

The messaging system is built on UDP (User Datagram Protocol), an unreliable, connectionless protocol for sending information on the Internet. The server and clients hence listen for UDP packets on the network, and perform their own decoding and parsing of the data received. Since reliability of message delivery was a desired feature, the developers of the messaging system had to implement it using UDP. It is perceived that using UDP and a proprietary data encoding is used for performance reasons [12].

This is an example of a system that was designed with performance in mind. If the server cannot efficiently send updates to the clients, the game play will appear discontinuous and "jerky" on the client machines. Although time constraints on the change notification facility we are building are not as strict, reliability of message delivery is [12].

2.2 Zephyr Notification Service

The Zephyr Notification Service is one of the fruits of the Project Athena distributed computing environment used at MIT. Zephyr is used to send text messages between users and groups of users on hosts within Athena. Users on Athena can send messages to each other, and can carry on conversations in real time. The messages appear in windows that "pop" up onto a user's screen without warning. A user can send a message to a named group of users, called an instance. A centralized server keeps track of who is online to receive messages, and who is a member of an instance [2].

Each host in the Athena computing environment runs a small client, called the Zephyr Host Manager (zhm) that is responsible for distributing received messages to users that have a session on that host. Each user on a host runs a Zephyr Window-Gram Client (zwgc), which handles displaying received messages from the zhm on the host. The

Zephyr server keeps track of which hosts a user is logged in to. When a user sends a message, it sends the message with a header that describes the target user or instance that the message should go to. The zhm for the host sends the information to the server as UDP packets [2].

The Zephyr server, receiving the message, takes different action based on the information in the header. If the target of the message is a user, it gets host information on the target user from its database. If the target of the message is an instance, the server gets a list of users that have subscribed to the instance, along with which host each user has a session on. The server then sends the message to the zhm of each host that should receive the message. In the case of sending a message to an instance, the message is sent to each host in the list. Once again, the message is sent as UDP packets. Since the underlying transport of messages is in the form of UDP packets, unreliable, out-of-order delivery can occur. The basic Zephyr Notification Service offers no reliability or ordering of messages. Applications using the service that require reliability or ordering need to implement these properties on top of the service [2].

The Zephyr Notification Service can be seen as a system that maintains conversations. When the conversation progresses, in the form of new messages being generated by users, the system asynchronously sends the messages to the intended users. The idea of a user subscribing to an instance of conversation is also valuable. This allows the user to register their own interest, in addition to having messages that are targeted to just them. The messages that are sent to the users characterize the change within the system. This change is sent to clients without the clients having to contact the servers for the information. The notion of asynchronous communication from the server to the client, even at this high level, is a desirable property in a change notification facility. We do not want the clients constantly waiting for information if there is none to give, but we want clients to be told as soon as possible when a change has been made [2].

2.3 MarketSheet

MarketSheet, a product developed by TIBCO Software, is used by securities traders to analyze securities that the trader is interested in. The graphical user interface for

MarketSheet lists stocks and their current price, along with information about the how the price has changed over time. In addition to just prices, MarketSheet includes a list of the latest headlines from news sources like the Associated Press. A trader uses the displayed information to make decisions about what to buy or sell over the course of the day. What makes MarketSheet special is that fact that it receives updated information in the form of asynchronous messages, and changes the graphical user interface based on the information it receives. MarketSheet will color the entry for a stock green if the messages it receives indicate that the price is increasing steadily, or red if the price is sharply declining [11].

The servers that MarketSheet depend on for this information translate data from various news sources, and transmit the information using a publish/subscribe architecture. The servers publish the data to particular subjects that the MarketSheet listens to. The messages are sent to the MarketSheet at the time they are published, producing the effect that the information comes in real-time. The MarketSheet is one of the first uses of the event-driven technologies developed by TIBCO, and is used at Fidelity Investments. TIBCO markets a package called TIB/Rendezvous, which allows developers to create applications that publish data and subscribe to subjects to receive that data on what TIBCO calls "The Information Bus" [11][8].

MarketSheet, and the system that supplies it with information, is the perfect example use of a change notification facility. The change information is pushed to the clients, and part of the MarketSheet program deals with that information as it flows from the publishing servers. What results is having the clients reflect exactly what information the server contains, and if any change should occur, clients are automatically notified [11].

2.4 Applicability to Workflow Management Systems

Looking at these three examples, we begin to see some features that are common to change notification facilities:

- The idea of clients subscribing to a group to receive a particular type of messages.

- The option of reliably or unreliably sending messages to clients, and whether or not order or receiving the messages is important.
- The importance of performance, through imposing a timed constraint on state updates, using a connectionless transport protocol such as UDP, or taking advantage of multicast communication.
- The idea that clients do not need to maintain the complete state of the server, only the state that the client is interested in.

Although many of the above techniques are applicable and are considered in this thesis, there are aspects of workflow management systems that require additional mechanisms. In particular, the model of data for workflow systems (such as InConcert) is more complex and interconnected than just a collection of stocks each of which has a price (and maybe other properties) but no relationship to any other stock. A typical workflow "task list" client (which we describe in detail in Chapter 6) presents to workflow users the set of tasks they need to work on, which is the result of one or more database queries on task objects and their current state and role assignment. Properly updating this information requires careful matching and processing of change notification events, such as updates to task state and assignment, to determine what effect, if any, they have on the results of the queries being displayed to the user.

Taking a look at systems that provide change information, and considering how they provide this information brings forth issues that we have to deal with in the design of the new facility. The next chapter discusses the design of the change notification facility and addresses many of these issues in detail.

Chapter 3: Design of the Change Notification Facility

The primary goal of this thesis is to provide a means for notifying clients when information has changed on the server. This chapter describes the components and features of this facility. We address the basic questions that need to be answered in order for this facility to be successfully built.

3.1 Information to Make Available

The information that is made available to the clients has to adequately describe the change that has occurred on the server. The information could involve a generic description of an event that has occurred, or could specifically state which object properties have changed for the InConcert objects, and what the value has changed to. There are events that are currently logged in the InConcert server, which don't necessarily describe which properties have changed, but offer an adequate description of what has changed on the server. For the thesis, we will use the current InConcert event as the information we send to clients [6].

The facility will use the existing InConcert event object for two reasons: familiarity and performance. Adding a new class for the information that is transmitted to the clients will add one more object to the InConcert object hierarchy. Client application programmers will have to understand the behavior of this "new" object, although it would look very much like an event. Having our own object for the information would allow us to define what data members we want, and possibly describe exactly which properties are changed, but the programmer using the service would have to become familiar with another object. We can leverage the previous knowledge of events, making it easier for a client application programmer to use the new facility.

Creating a new class to handle the information that is sent out to clients will also add more architecture to the InConcert server. Creation and storage of the new class could conceivably involve adding a new database table, so instances of the new class can be logged for distributing to clients. This addition of a database table and functions to access it will be unnecessary if we simply use the existing event.

If necessary, we can expand the InConcert event class (known specifically as IcEvent) to contain more data, and create new event types to characterize new functionality in the server. For example, there are currently no events that detail the setting of an attribute to a new value. These event types will be added so that the change notification facility can notify clients when this occurs. Chapter 6 details information that was added to some existing event types so that the event was more helpful to the demonstration application that was written to use the facility.

3.2 Message Transport

InConcert client applications can currently ask the server for InConcert events of a particular type, or events associated with a particular object that have occurred over a particular interval of time. The fact that the events need to be asked for raises the question of what benefits could be obtained if these events where delivered without asking.

3.2.1 Polling or Pushing?

For a client to periodically poll for the recent events runs the risk of asking for events when there are none to give. In addition, the polling can result in unnecessary load on the server if many clients are polling for events. There is an alternative to polling, that involves having the server send the event to the clients within a reasonable time after the event occurs. This is commonly referred to as pushing. The design decision to not push the events could have been made long ago, when there was not enough time or processor and network resources to build such a facility. Now that computers are faster, network bandwidth is higher, and programming languages allow multi-threading (which is ideal for handling asynchronous events), the issue of whether or not to push events to each client needs to be explored [9][10].

3.2.2 Unicast or Multicast?

In order to notify clients using a push model, information has to be sent from the place where the information is stored (the InConcert server) to the place where notification is

needed (an InConcert client). There are two general ways of doing this. One is to send the same message to each client. This requires sending the message n times for n clients. Another way is to have the server send out one message to a group, multicasting the message to any client who listens to that group.

Sending the same message to each client, known as unicast, requires knowledge about which clients wanting to receive messages. This would require some record keeping as to which clients are interested, including having the clients explicitly register with the server that is sending messages. It is easy to see that this method wastes network bandwidth and decreases server performance by sending the same message multiple times, as many times as there are clients listening.

Multicasting messages from a server does not involve sending a message directly to each client host, but sending a message to a group of client hosts. Multicast architectures have publishers that send messages, and subscribers, which receive messages. In the top level of design, the publishers do not need to keep track of subscribers that will receive messages. A multicast architecture could be considered an abstraction of unicasting to a group of interested clients, and the only ideas that are exposed are a common location for message sending and receiving, and the idea of publishing and subscribing to this common location. However, this abstraction allows for optimizations in which a message is not sent n times to n clients. There are hardware and software implementations of the multicast architecture, and some with a combination of hardware and software [1][3][7].

Since a multicasting architecture does not care about how many clients are listening, there is no need to expose information dealing with clients in the publishing part of the abstraction. Adding a multicast package to the InConcert server will therefore require minimal additional design. There will be additional design in the initialization of the server, to start the publisher, and additional design to publish each event. The performance of the InConcert server will not be affected drastically because it needs to perform only one additional action (publish) per event.

3.2.3 Clients receive every event in the order that they occur

A vital property of the information that we send is that the server sends the events in the order that they occur within the server, and that the clients receive the events in the order in which they were sent. The result of these two facts is that the clients receive the events in the order that they occur. In more technical terms, the client must receive the events in First In, First Out (FIFO) order. The necessity of a FIFO ordering of messages is made clear with a simple example. A client could receive two events that denote that the same property has changed to two differing values, Value1 for the first event and Value2 for the second. The client, after processing each event in the order of arrival, finally sets the value of the property locally to the value in the last event that it processes. The client could improperly set the value if the events were sent to the client out of order (if, for instance, Event 2 occurs in the server before Event 1).

Another vital property is that the clients receive every event that is published while the client is listening. This property will require the multicast architecture to reliably deliver the events to clients. Some multicast packages do not ensure reliability of message delivery. This is because some applications, like multicasting audio and video, do not require that every message is delivered to the client, just enough to produce a decent picture or sound. In the case of the change notification facility, every message is necessary because missing a message means that the client has missed information. Missing messages is not desirable because it defeats the original goal of having the clients better reflect the state of the server. The client has to have the opportunity to take action on every event that occurs in the server [10].

3.2.4 TIB/Rendezvous

For this thesis, we will use a reliable multicast transport protocol for sending events to clients. This will allow us to focus on the details of how each client interprets the events, instead of focusing on how they are transported. For this thesis, the events will be transported using a commercial product called TIB/Rendezvous [13].

TIB/Rendezvous is a product developed by TIBCO Software Incorporated. Rendezvous provides application level reliable multicasting of messages to clients. Rendezvous follows the publish/subscribe architecture for multicasting messages, in addition to

providing support for the request/reply architecture that is common to clients and servers. The API provided by the Rendezvous libraries abstract away many of the details of message delivery, and allow the developer to focus on the content and structure of the messages. Multicasting is done to particular subject names, where a client can subscribe to the subject and receive messages. Rendezvous subjects are based on a hierarchical naming structure, where periods separate the elements of the subject description. The data structure for containing a message in TIB/Rendezvous is a Rendezvous message. This data structure can contain a collection of strings, integers, dates, binary and encrypted data, and can also include other Rendezvous messages. Each value is keyed with a string within the Rendezvous message. The maximum size of a single Rendezvous message is 64 Megabytes, well above the limit for our purpose [13].

Rendezvous depends on each publisher and subscriber either running the Rendezvous daemon (rvd), or being connected to a Rendezvous daemon by a Rendezvous agent (rva). The Rendezvous daemon handles all of the reliability and ordering of messages that enables FIFO delivery. Client applications that wish to use Rendezvous do so by creating a Rendezvous session. Using the session, the clients can create Rendezvous senders to publish messages, or Rendezvous listeners to subscribe to subjects and receive messages [13].

The parameters that are needed for starting a Rendezvous session include a Network, Service, and Daemon parameter. The Network parameter describes which IP port the Rendezvous daemon will listen to on behalf of the session. The Service parameter specifies which network interface will be used by the session, if there is more than on installed on the host. The Service parameter also allows the use of multicast group addressing for the transport of messages, taking advantage of network hardware enhancements that support multicasting. Using the Service parameter, you can specify which multicast groups to listen to for messages, and which group is used to send messages. The version of TIB/Rendezvous used for the work on this thesis is version 5.0 [13].

For this thesis, we will publish all events to a single subject that clients can listen to. The name of the subject is dependent on the name of the InConcert server. If the name of the server is "budget", the subject the clients listen to will be "`com.inconcert.budget.eventservice`". Further discussion on publishing to different subjects depending on the type of event will be discussed in later chapters.

3.3 Presenting the Information to clients

Clients can use the information published by the server in a variety of ways. For now, the goal is to provide an adequate interface to allow the maximum amount of uses of the facility. The target language for the client interface will be Java. Java was chosen because it is an increasingly popular and portable language. Java also cleanly provides facilities for multithreading, which is necessary to receive information from the network in an asynchronous fashion like our design mandates.

The client interface should go along side the current InConcert Java Application Programming Interface, known as the IcJava API. Programmers who are familiar with the IcJava API should be able to use the facility to enhance their applications. The interface should allow multiple objects within the client to listen for the event messages, much like Java objects listen for events from Swing GUI components. This interaction among the facility and the objects is commonly known as the Observer design pattern [4]. In addition, a way to filter the events will be provided, so listeners can easily discriminate between the events they are interested in and the ones they are not.

Of course the facility will provide the event information to the Java clients in the form of InConcert events, namely instances of the IcJava class IcEvent. The main class that will provide the functionality of delegating the IcEvents will be called the IcEventService. Please note that from the client's point of view, the change notification facility is called the IcEventService. The names can be used interchangeably.

3.4 Change Notification Facility Design Summary

The addition of the change notification facility will be accomplished by extending the functionality of both the server and client. It is important to show the relationship between the extensions that are being added. Figure 1 shows the relationship between the server that now publishes events, and the clients that subscribe and listen for them. The darkened arrows denote the flow of a published event.

The host on the bottom is running the InConcert server named "`finance`". Within the server, the Event Publisher encodes the events that are created and publishes them as Rendezvous messages. Each Rendezvous message that is published by the server is given to the Rendezvous daemon, which handles the publishing of the message to the subject "`com.inconcert.finance.eventservice`".

Each of the three hosts at the top of the figure is running an InConcert client that uses the change notification facility. Each client has an instance of IcEventService that is created to listen for the events that are published by the InConcert server "`finance`". The Rendezvous daemon within each client is subscribed to the Rendezvous subject "`com.inconcert.finance.eventservice`", and passes the Rendezvous messages that it receives to the IcEventService. Upon receiving the Rendezvous message, the IcEventService decodes it into an IcEvent, and passes the event to the Listeners that are registered with the IcEventService.

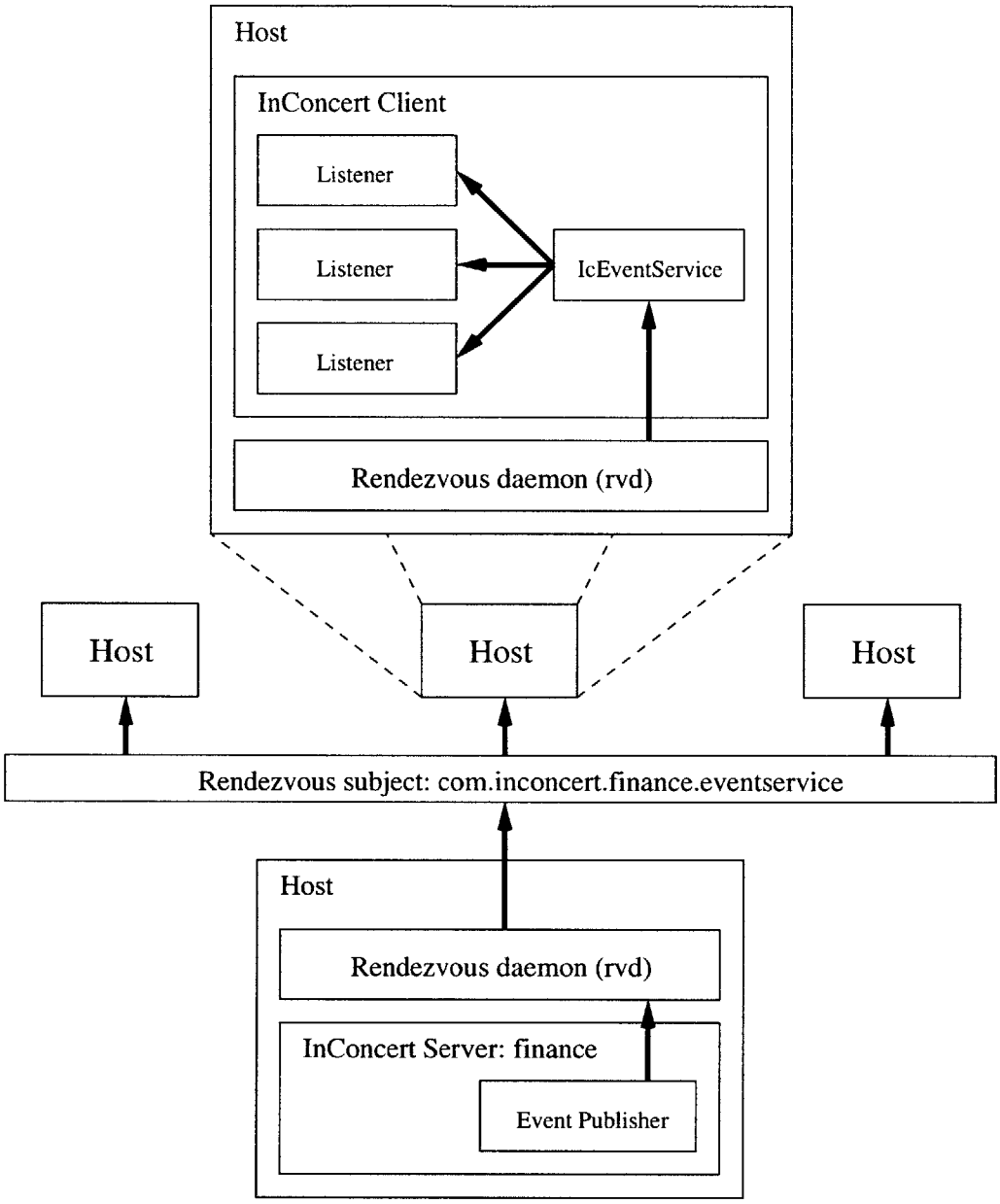

Figure 1: The relationship between the client and server extensions

The two following chapters describe the client and server extensions, and how they were added to the existing client/server architecture.

Chapter 4: The Change Notification Facility Server Extension

The first step of building this facility is to construct a way for the server to publish the events that it accumulates over the course of operation. In order to add the functionality of publishing the events to clients, it is necessary to understand the underlying architecture of the InConcert server, and how it normally processes these events.

4.1 The InConcert Event/Action Model

The InConcert server can be considered as a storage location for the workflow data. When this data has changed, InConcert internally logs an event detailing the change. InConcert has a wide range of event types that are logged internally within the server, and ultimately written to a database table and optionally a log file. Examples of different types of events include a completed job, an acquired task, a document being checked out, and a user being added to a pool.

In addition to event logging, InConcert server offers the capability to perform a particular action whenever a specific event happens on the server. These actions can be established by client applications with the use of triggers. A trigger is a named pair containing an event specification (EventSpec), and an action specification (ActionSpec). The EventSpec is a description of the event that the trigger is triggered on. The ActionSpec is the description of the action that is performed when the event occurs. An action can entail creating a job instance from a template, sending an electronic mail message, or performing a Remote Procedure Call (RPC) to a host [6]. This Event/Action Model is a powerful feature for InConcert, and allows open-ended functionality for systems developers that want to integrate InConcert into their company's existing computing environment.

4.2 Architecture of the InConcert server

The InConcert server can be described architecturally as a system that maintains a database of workflow information and listens to requests to retrieve and change that information. The server is written in C++, so any extensions written for the server need to be written in C++. The InConcert server creates the abstraction of the workflow

objects and their behavior using database tables and queries. The current UNIX version of InConcert can use an Oracle or Sybase database to store its information. Clients that access InConcert use the Applications Programming Interface (API) provided with the InConcert server [5].

The server consists of three types of processes that run concurrently as the server. The relationship between the processes and the database is visualized in Figure 2. The first type is the Worker process. A Worker process takes an API call from the client and performs the particular work that is requested. Each Worker process listens for RPC calls that correspond to the API calls that are made by clients. Each API call has a unique program number that the Worker process uses in order to know what work to perform. The work performed by the Worker process could involve retrieving data from the database or changing data in the database, and afterward logging the event corresponding to what has happened during the call. The Worker process returns to the client the information retrieved along with status information on whether the call was successful. The InConcert server can run multiple Worker processes, the number specified at the time the server is started [5].

The second type of process is the Dispatcher. The Dispatcher makes sure the Worker processes start up properly, in addition to distributing client connections across the worker processes. When a client connects to the InConcert server, it first makes a connection with the Dispatcher process. The Dispatcher provides the client with the particular information to connect to a Worker process. During a session of InConcert, a client is connected to the same Worker process [5].

The third type of process is the Event/Action daemon. The Event/Action daemon periodically looks in a table of new events, and fires the triggers that match the particular events that have occurred. For every event, it locates which triggers have a matching EventSpec. And when a match occurs, the ActionSpec associated with that EventSpec is performed, thereby firing the trigger. Only one Event/Action daemon can be run per InConcert server [5].

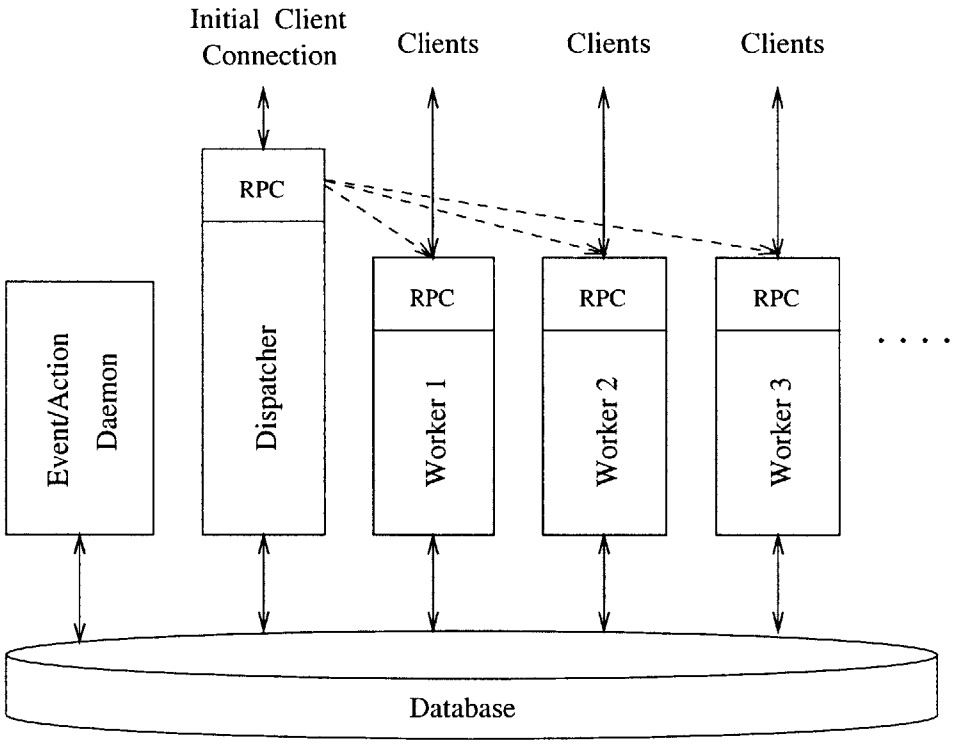

Figure 2: InConcert server Architecture

4.3 Implementation of the Server Extension

The main functionality that we want to add to the server is the ability to publish messages. Specifically, we want the server to publish the information that is contained in the events (instances of the class IcEvent) that are created and logged by the Event/Action daemon.

A good way to do this would be to add a class whose sole purpose is to take an IcEvent, encode it in whatever format we wish, and multicast the message. The class would require initialization parameters to specify how messages are going to be published. The set of initialization parameters will be specific to the underlying multicast transport protocol, in this case TIB/Rendezvous. We could design a class that properly abstracts the details of the message transport, which could allow us to change the transport protocol easily. We would only have to replace the class with one that implements the same interface.

The class developed for this purpose is called the IcEventNotifier. It uses a separate helper class for encoding the IcEvents, called the RvIcEventEncoder.

4.3.1 IcEventNotifier Class Interface Summary

```
IcEventNotifier::IcEventNotifier( const IcString& icServerName,
                                  const IcString& rvService,
                                  const IcString& rvNetwork,
                                  const IcString& rvDaemon )
```

This constructor constructs an IcEventNotifier given the InConcert server name `icServerName`, and the three Rendezvous session initialization parameters `rvService`, `rvNetwork`, and `rvDaemon`. Within the constructor, a session with the Rendezvous daemon is established. Given the InConcert server name, it builds the subject that is used to publish the Rendezvous messages. For example, given the server name "`ulysses`", the IcEventNotifier publishes events to the Rendezvous subject: "`com.inconcert.ulysses.eventservice`". This design allows a multiple InConcert server environment, where each server would be publishing events to a different subject, based on their server name. A Rendezvous sender (RvSender) is created with the generated subject name and is kept in memory to send events.

```
IcStatus IcEventNotifier::notify( const IcEvent& event )
```

This method is used by the InConcert server to encode and publish an IcEvent. It simply uses the static methods in the RvIcEventEncoder class to create a Rendezvous message (instance of the class RvMsg) from the event and gets the RvSender to publish the RvMsg.

4.3.2 Rendezvous message encoding of an IcEvent

The RvIcEventEncoder class takes an IcEvent and encodes its data into a Rendezvous message. Since information in the IcEvent differs on the type of event, this helper class does something different depending on the event type, in order to extract the correct information from the IcEvent. It is necessary to point out that this class hides a great deal of the Rendezvous details of encoding an IcEvent, and could be replaced easily with another helper class that encodes an IcEvent in another particular data format.

IcEvents contain common data, along with data that is specific to the event type. The following table describes the data members of an InConcert event [6].

Data Member	Type	Description
Id	String	The database id of the event
Name	String	Name of the event
Type	Integer	The type of the event
Time	DateTime	the date the event was created
Creator	User	The InConcert object denoting the user that created the event
CreatorName	String	Name of the user that created the event
Translation	String	A string that describes the specifics of the event
Object1	Object	Event specific data, containing InConcert objects and names of objects that are specific to the event, along with a general purpose Integer that is called "Version". This Integer commonly holds the version of the document when the event type deals with document manipulation.
Object1Name	String	
Object2	Object	
Object2Name	String	
Object3	Object	
Object3Name	String	
Version	Integer	

Table 1: Data members of standard InConcert event

An InConcert event is encoded according to the following table. The top level structure of the encoding is a RvMsg, and contains the following information:

Field Name	Field Type	Description
IcEvent_Id	String	The InConcert Id for the IcEvent
IcEvent_Name	String	The name of the event
IcEvent_Type	Integer	The event type of the IcEvent
IcEvent_Time	Date	The time that the IcEvent was created.
IcEvent_Creator	String	The InConcert Id of the User that created the IcEvent.
IcEvent_Creator_Name	String	The name of the User that created the IcEvent
IcEvent_Translation	String	A complete description of the event, including event-type specific information.
IcEvent_Data	RvMsg	A Rendezvous message that contains the event-type specific information of the IcEvent

Table 2: Fields within Rendezvous message encoding an IcEvent

Note that IcEvent_Data is a nested Rendezvous message that contains the event-type specific information, and is encoded according to the following table:

Field Name	Field Type	Description
Object1	RvMsg	The first encoded InConcert object
Object1_Name	String	The name of the InConcert object Object1
Object2	RvMsg	The second encoded InConcert object
Object2_Name	String	The name of the InConcert object Object2
Object3	RvMsg	The third encoded InConcert object
Object3_Name	String	The name of the InConcert object Object3
Version	Integer	An Integer that is used to contain many different kinds of information, it is named Version because it is more often used to contain a version number of an InConcert Document object

Table 3: Fields within the Rendezvous message IcEvent_Data

Object1, Object2, and Object3 are InConcert primary objects, and are encoded according to the following table.

Field Name	Field Type	Description
Id	String	The Id of the InConcert object
Class_Branch	String	A string describing the primary class branch of the InConcert object. The value of the string can be "Job", "Task", "Role", "Binder", "User", "Pool", "Document", "Link", "Repository", "Trigger", "Class", "CustomObject", or "None".

Table 4: Fields within the Rendezvous message encoding an InConcert object

4.3.3 Publishing options: One or many subjects?

It is conceivable that we could publish the events in a way that could make it easier to filter. We could publish each event type to its own subject within the "eventservice" hierarchy. However, since there are on the order of 80 different event types, organizing the listener to listen to a set of subjects could be cumbersome [6]. For now we will publish to only one subject, and create our client extension accordingly.

4.3.4 RvIcEventEncoder Class Interface Summary

```
public static void RvIcEventEncoder::encodeIcEvent( RvMsg& msg,
                                                     RvSession& rvs,
```

```
                                                    const IcEvent& evt )
```

This method performs the encoding of the InConcert event and is called by the IcEventNotifier. This method encodes common and event type specific data using other helper methods within the class. The method modifies the RvMsg `msg`, adding the common event data to the Rendezvous message. The event type specific data is stored in a separate RvMsg and added to the `msg`.

```
private static void RvIcEventEncoder::encodeIcObject( RvMsg& msg,
                                                       const char* name,
                                                       IcObject obj )

private static void RvIcEventEncoder::encodeIcClassBranch( RvMsg& msg,
                                                           const IcObject& obj )

private static void RvIcEventEncoder::encodeIcString( RvMsg& msg,
                                                      const char* name,
                                                      const IcString& str )

private static void RvIcEventEncoder::encodeIcInteger( RvMsg& msg,
                                                       const char* name,
                                                       const IcInteger& igr )

private static void RvIcEventEncoder::encodeIcPrivilege( RvMsg& msg,
                                                         const char* name,
                                                         const IcPrivilege& prv)

private static void RvIcEventEncoder::encodeIcDateTime( RvMsg& msg,
                                                        RvSession& rvs,
                                                        const char* name,
                                                        const IcDateTime& dte )
```

These methods are used to encode InConcert data types into a common format within a Rendezvous message. These methods take the RvMsg that they are going to alter, the name that they are going to give the value that they are adding, and the data that they are going to encode. In order to encode an InConcert DateTime object, a RvSession needs to be passed as an argument because the Rendezvous session contains platform specific information about how date and time information are encoded.

4.4 Configuration hooks for the IcEventNotifier in the InConcert server

Parameters have been provided so that the administrator can correctly configure the server to publish events given the network environment. The three parameters that govern the creation of a Rendezvous session (Service, Network, Daemon) were added to

the InConcert server parameters as IC_RV_NOTIFIER_SERVICE, IC_RV_NOTIFIER_NETWORK, and IC_RV_NOTIFIER_DAEMON. The configuration parameters can be altered during the startup of the InConcert server. The parameters, once read at startup, are static strings within the server and can be passed to the IcEventNotifier constructor along with the InConcert server name.

4.5 Placement of the Server Extension

The IcEventNotifier class needs to be integrated into the server in a way that it has access to and publishes every IcEvent generated by the InConcert server. There are two general times when the IcEvents can be published using the IcEventNotifier. Each time alludes to a different strategy to place the extension within the server.

4.5.1 Strategy 1: Publish the events before they are permanently logged

This strategy mandates that the IcEvents are published while they are being permanently logged to the database, which occurs during the Event/Action daemon polling loop. This would require the Event/Action daemon to contain an instance of IcEventNotifier, and use the instance to publish each event. The publishing of the event will happen before triggers on the event are processed and the event is written to the database log. This strategy yields a simple solution, and was the first approach thought of in adding the server extension to the InConcert server.

The fact that the Event/Action daemon works by a polling loop, by which it sleeps for a period of time and is reactivated, puts a constraint on the timeliness that IcEvents are published. The amount of time that the Event/Action daemon sleeps is configurable by the administrator of the InConcert server, so a more timely publishing of IcEvents is possible. The default time to sleep for the Event/Action daemon is 60 seconds [5].

4.5.2 Strategy 2: Publish the events the instant after the event is created

This strategy mandates that the IcEvents are published immediately after the API call that created the events has been handled by the worker process. Each Worker process would have an instance of IcEventNotifier, and publish the events after the API call has finished

handling the RPC call from the client. IcEvents are also created by the Event/Action daemon, when it processes event and "fires" triggers based on the events it processes. Therefore the Event/Action daemon will also require an instance of IcEventNotifier to publish the events related to triggers.

One downside to strategy 2 is the fact that a Worker process could die between performing an API call and publishing the event. In this situation, the change would be committed to the server, but no event would be published, and this would not be ideal. Another downside is that strategy 2 requires more integration within the InConcert server, requiring an IcEventNotifier per Worker process and one for the Event/Action daemon [5].

The implementation of the InConcert server with the server extension uses the design suggested by strategy 1. An instance of IcEventNotifier is created in the Event/Action daemon, and before the daemon processes the new events for triggers that could be set on them, the daemon iterates through each event and publishes it with the IcEventNotifier.

4.6 Running and building the InConcert server with TIB/Rendezvous

Now that the InConcert server has been extended to use TIB/Rendezvous, additional environment settings are needed to allow the InConcert server to function. The PATH environment variable needs to include the directory that contains the Rendezvous daemon (rvd) when running the InConcert server. This is necessary because when an instance of a RvSession is constructed, a Rendezvous daemon needs to be started on the host if one is not currently running. The C++ version of the TIB/Rendezvous API has the ability to start a Rendezvous daemon if one is not running. The C++ TIB/Rendezvous API uses the PATH variable to locate the Rendezvous daemon application (rvd) in order to start it [14].

In the development process of adding the change notification facility extension to the InConcert server, the appropriate C++ Rendezvous libraries were added to the makefiles for the server components that directly used the IcEventNotifier and RvIcEventEncoder.

The source code for the extension classes was added to the source tree for a version 4.5 of the InConcert server.

The server extension is useless without giving clients a way to access the events that are published. The next chapter describes the client interface implemented for this thesis.

Chapter 5: The Change Notification Facility Client Extension

The client extension of the change notification facility provides a simple interface that allows a Java programmer to use the facility along with the InConcert Java Application Programming Interface (IcJava API). Part of the structure of the interface is similar to the Java event/listener model that is used to distribute events generated by the Graphical User Interface to application objects that anticipate user input. Designing the interface in this way will allow Java developers to easily adapt to using the new service. They will be able to program applications that behave almost as if the client is generating the InConcert events itself, instead of receiving them from a remote host.

5.1 Implementation of Client Extension

The client interface consists of two classes and two interfaces. The two classes provide the implementation of the service in terms of listening and decoding of Rendezvous messages into IcEvents. The two interfaces provide application programmers with a way to access the service. The classes and interfaces will be added to the InConcert Java distribution, and are located in Java class package "`com.inconcert.eventservice`".

5.1.1 IcEventService Class Interface Summary

The IcEventService class provides the application's connection to the event service. The constructor for IcEventService takes the parameters that are used to create a Rendezvous session, along with the name of the InConcert server that the service should listen to. Given the name of the server, the IcEventService creates a Rendezvous listener, that listens for messages on the subject: "`com.inconcert.<servername>.eventservice`", where `<servername>` is the InConcert server name. This is the same subject that the InConcert server publishes messages to. The IcEventService object maintains a list of objects that have registered to listen for events from the service. When an event is received, the IcEventService object delegates the event to each listener in the list. The object uses a Java inner class IcEventServiceCallback, which simply implements the TIB/Rendezvous RvServiceCallback interface and performs the decoding and delegation of the Rendevous message when it is received [16].

```
public IcEventService( String server,
                       String service,
                       String network,
                       String daemon ) throws IcException
```

This constructor for the IcEventService uses the Server/Network/Daemon configuration of TIB/Rendezvous. Given the InConcert server name and these three parameters, the IcEventService is constructed and starts listening for events. An IcException is thrown when a Rendezvous session cannot be constructed, or when a listener cannot be added to the Rendezvous session. Similar constructors for the IcEventService that use the Server/Network/Daemon configuration are supplied for constructing the service with default parameters.

```
public IcEventService( String server,
                       String hostname,
                       int port ) throws IcException
```

This constructor for the IcEventService uses the Hostname/Port configuration of TIB/Rendezvous. Given the InConcert server name and these two parameters, the IcEventService is constructed and starts listening for messages. An IcException is thrown when a Rendezvous session cannot be constructed, or when a listener cannot be added to the Rendezvous session. Similar constructors for the IcEventService that use the Host/Port configuration are supplied for constructing the service with default information. This constructor is provided for Java applets that cannot start a Rendezvous daemon on their host computer, and need to connect to a daemon through another host that is running a Rendezvous agent.

```
public void stopService() throws IcException
```

This method stops the underlying TIB/Rendezvous service from listening for messages. An IcException is thrown when there is a problem stoping the underlying Rendezvous session.

```
public synchronized void addListener( IcEventListener evtL )
public synchronized void removeListener( IcEventListener evtL )
```

The first method is used to add listeners to the IcEventService. When a message is received, it is delegated to each of the listeners that have been added to the service using this method. The method is synchronized to atomically add the listener, so the delegation of a received event cannot occur during the addition of a listener. The method will not add the listener if it is already in the list. The second method removes a listener from the IcEventService, so it no longer receives events.

5.1.2 IcEventFactory Class Interface Summary

The IcEventFactory class is used by the IcEventService class to convert a Rendezvous message to a IcEvent object.

```
public static IcEvent parseEvent( Object message )
```

This method, the only functionality for this class, is a static method that takes a Rendezvous message and attempts to construct an IcEvent from the contained information. It uses the specified encoding used by the InConcert server to decode `message` into an IcEvent. If a failure occurs at any point in the decoding (due to bad formatting of the message), a null object reference is returned. The idea to have `null` returned, instead of throwing an exception upon error was chosen in order to increase the efficiency of the parsing. Throwing exceptions has been known to be inefficient in terms of time and space, and testing for the value of `null` returned by `parseEvent` can be done easily.

5.1.3 IcEventListener Interface Summary

The IcEventListener interface defines the functionality that an object needs to implement in order to have it receive IcEvents from the service. There is only one method in the interface.

```
public void receiveEvent( IcEvent evt )
```

This method, the only required functionality of an object that implements the IcEventListener interface, is called by the IcEventService in order to delegate the IcEvent to the object that implements this interface.

5.1.4 IcEventFilter Interface Summary

The IcEventFilter interface suggests functionality that can be used in order to make it easier for an object implementing the IcEventListener interface to determine whether it is interested in the IcEvent that was delegated to it. Although usage of an IcEventFilter is not enforced, it is provided as an aid to make determining event interest easier. IcEventEditableFilter, an example implementation of the IcEventFilter interface that filters events according to their type, is provided with the client package.

```
public boolean apply( IcEvent evt )
```

This method applies the filter to the IcEvent, returning true if the IcEventFilter passes the IcEvent, and returning false if the IcEventFilter blocks the IcEvent.

Although the IcEventFilter interface is simple, the objects that implement this interface can be extremely powerful. Filters can be created to discriminate between IcEvents based on event type, who created the event, when the event was created, or any data contained in the IcEvent object.

Now that the client and server extensions are developed and in place, it would be good to put the change notification facility to use. The following chapter details the design of an application that takes advantage of the change notification facility.

Chapter 6: Demonstration client that uses the Change Notification Facility

This chapter details the design of a client that uses the change notification facility. An overview of how InConcert manages workflow is presented, along with how the client specifically uses the change notification facility to better reflect the information on the server.

6.1 How InConcert manages workflow

InConcert solves the problem of workflow management using a task-oriented model of processes. InConcert organizes workflow in terms of processes (also known as jobs). The process is composed of a task dependency graph. Beginning at the start task, the task dependency graph details the order in which the tasks are to be performed. Branches in the workflow can occur, upon which there are conditions that govern which task should be performed next. Tasks can have documents attached to them, detailing the work that needs to be performed. Users of InConcert are organized into groups, called pools. A user can be a member of multiple pools. Ready tasks can be acquired and completed by any user in the assigned pool.

To make it easier to create processes, a user can create a process definition, which acts as a template that can create process instances (active processes). A process definition abstractly describes a process, not assigning a task to a particular pool of users, but to a particular role that has the necessary skill set. A process definition also allows the ability to include document binders, which are open references to documents. When the process definition is turned into an active process, the specific bindings of pools to roles and documents to binders can be made. The creation of process definitions allows a user to create their own library of particular types of processes, and can instantiate the definition on demand [6].

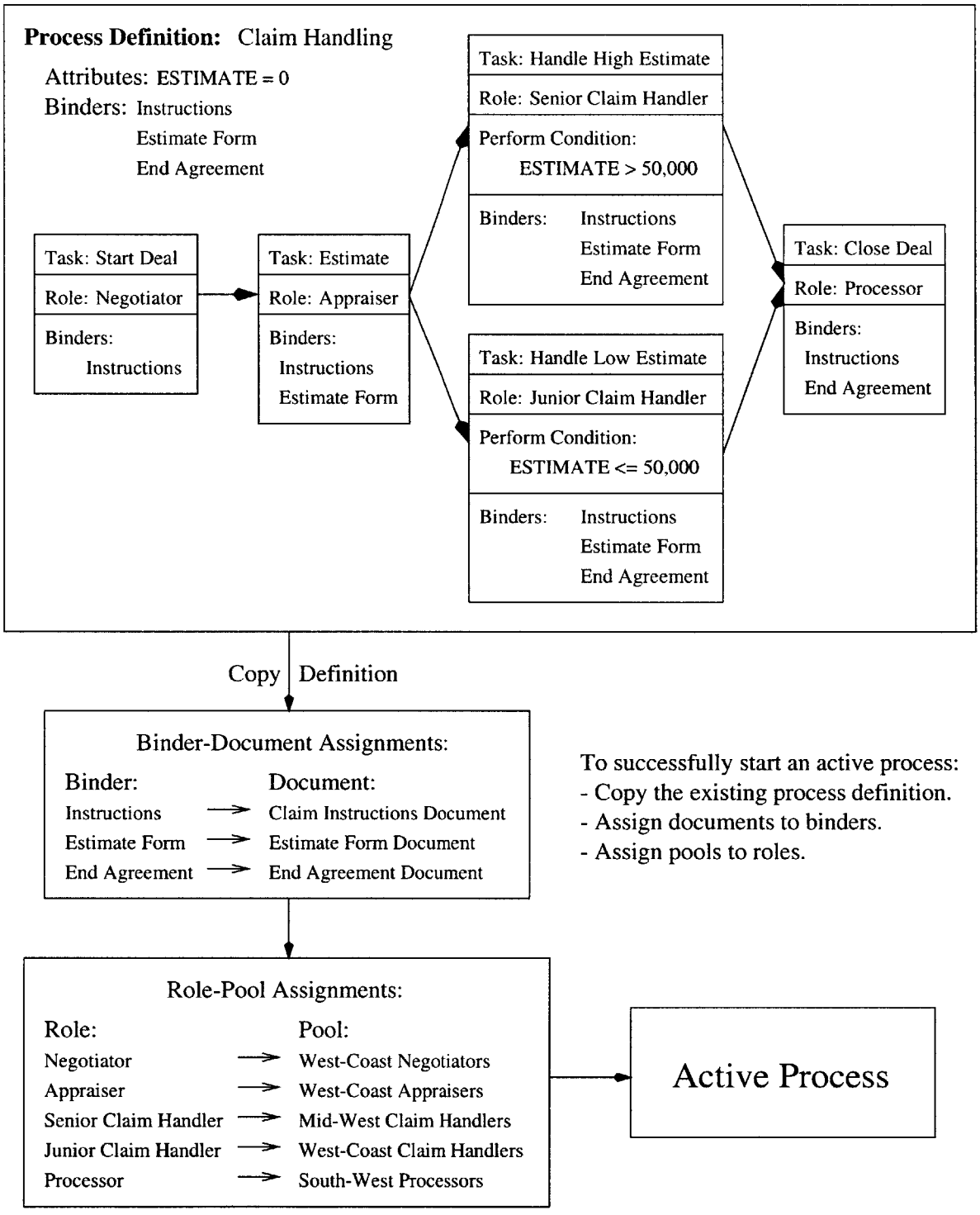

Figure 3: Workflow, in terms of InConcert objects.

The process definition diagrammed in Figure 3 details the process of handling an insurance claim. Each task has a role assigned to it that defines the skills that are required for the task. In order to have the process perform different tasks based on data

within the process, perform conditions are assigned to two tasks, based on the value of ESTIMATE. The claim will be handled by a more experienced person based on the value of the estimate set by the appraiser. The binders for the process include placeholders for an instruction document, estimate form and the end agreement. Starting an active process requires copying the definition, assigning pools to roles, and assigning documents to the binders.

6.2 ActiveTaskList, a real-time TaskList

The TaskList is the "killer application" of workflow management systems. The TaskList presents the user with the ready tasks that they have access to acquire, and tasks that they have acquired. Users can mark the acquired tasks as complete, and acquire more tasks from the list. The TaskList keeps track of work that the user can perform, and the work that they have elected to perform. The TaskList downloads the documents that are linked to the particular task that they have acquired, so the user can read and manipulate them according to the task at hand. Given the API that InConcert provides to access the server, the TaskList can take many forms. However, the functionality described above is common among workflow clients that are written to use InConcert [5].

The common TaskList can be enhanced by using the change notification facility. The events that are published by the InConcert server can provide timely information for a TaskList, such that it can update the information contained in the TaskList without making a query to the server.

A good example of using the change notification facility is having the TaskList listen for the published event TASK_READY. A TASK_READY event is created whenever a task becomes ready for acquisition by a pool of users. The information provided with the event is the task that has become ready, the job that the task belongs to, and the pool that the task is assigned to. The TaskList could listen for this event using the IcEventService, and upon receiving the event, determine whether the task was ready for any pool that the user was a member of. If the task's role was assigned to a pool that the user was a member of, then the user could acquire and perform that task. The TaskList application could add the task to the list of ready tasks for the user, updating the graphical user

interface so the user can see the task that can be acquired. If the task's role was assigned to a pool that the user was not a member of, then there would be no reason for the user to be interested in the task, and the TaskList application would do nothing. Ideally, if the TaskList could listen to the events published by the InConcert server, the information on the TaskList would always be up to date (except, of course, for communication latency). The user would always have an accurate view of what tasks are available for them.

The ActiveTaskList application is a Tasklist that listens to all pertinent events that are published by the InConcert server. The application shows tasks that are ready for the user, along with the tasks that the user has acquired. The user has the ability to select and acquire a ready task, and complete, release, and open an acquired task. In a more robust TaskList, opening an acquired task would give the ability to open and manipulate the documents and attributes that are associated with the task. However, to focus on the event-listening abilities of the ActiveTaskList, which demonstrates the value of the change notification facility, the "open" functionality was not implemented in the application.

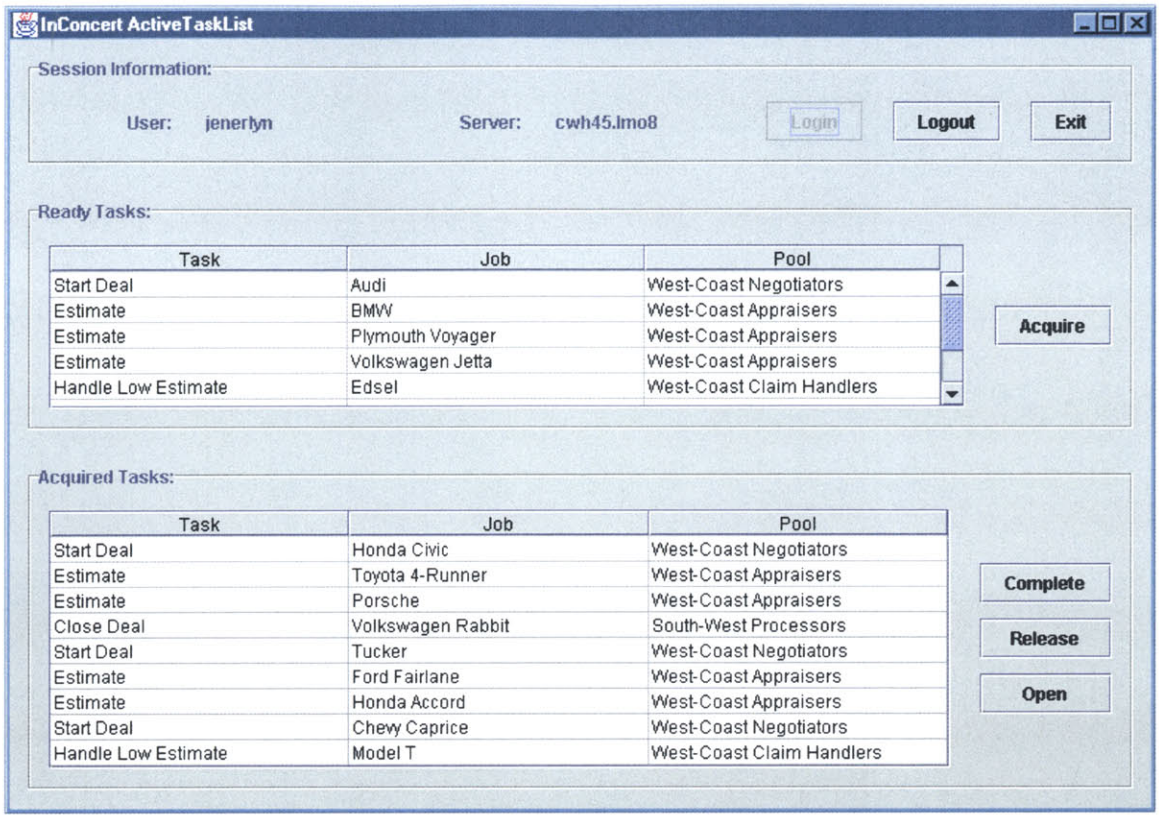

Figure 4: InConcert ActiveTaskList

Figure 4 is a screenshot of the ActiveTaskList. When a user logs in to the InConcert server, the client starts listening to the change notification facility. The client then initializes the interface by getting the user's lists of the ready and acquired tasks. When an event that is of interest to the ActiveTaskList is received, the application re-computes the contents of the lists of ready and acquired tasks. The handling of the events from the event service is the most interesting part of the application. Each important event type evokes some action by the application. The application maintains the following state and performs operations on it according to the events that it receives.

Name	Description
CurrentUser	The user that is logged in and is currently using the ActiveTaskList
UserPools	The set of pools that the user is a member of. This is kept to efficiently determine whether events that deal with tasks are important to the user.
ReadyTasksData[Task, Job, Role, Pool]	A set containing tasks and information on the tasks available for the user to acquire. The additional information includes the job the task is in, the role assigned to the task, and the pool assigned to the role.
AcquiredTasksData[Task, Job, Role, Pool]	A set containing tasks and information on the tasks the user has acquired. The additional information includes the job the task is in, the role assigned to the task, and the pool assigned to the role.

Table 5: State maintained by ActiveTaskList application

6.2.1 Adding more information to the published events

The following event types had the pool of the particular task added to the event information that is published, in order to make the ActiveTaskList perform better. These additions can be seen as an overall improvement of the event, putting the event described in a more defined context.

- TASK_ACTIVATE
- TASK_ACQUIRE
- TASK_COMPLETE
- TASK_LATE_START
- TASK_OVERDUE
- TASK_READY
- TASK_RELEASE
- TASK_SKIP
- TASK_WAITING

6.2.2 Actions taken by ActiveTaskList in response to events

The following event types are listened for by the ActiveTaskList application. Next to the event type name is the information that is provided with the event. Each event type listed

contains a description of the event, and what action the ActiveTaskList takes upon receiving the event [6].

- **TASK_ACQUIRE**(Task A, Job B, Pool C, User D)

  Description: This event occurs when user D has acquired the task A, which was assigned to pool C within job B.

  Action: If the task A is in ReadyTasksData, remove it. If the user D is equal to CurrentUser, then add the task A to AcquiredTasksData.

- **TASK_ACTIVATE**(Task A, Job B, Pool C)

  Description: This event occurs when task A, within job B, has been activated. This is a special feature, where by task A can be set to automatically activate when it has become ready. When task A is activated, the subtasks of A become ready. Essentially, when a task has become activated, it is neither ready or acquired, and should be removed from any TaskList.

  Action: If task A is in ReadyTasksData or AcquiredTasksData, remove it from that list.

- **TASK_COMPLETE**(Task A, Job B, Pool C)

  Description: This event occurs when a user has completed the task A within job B.

  Action: If task A is in ReadyTasksData or AcquiredTasksData, remove it from that list.

- **TASK_DELETE**(Task A, Job B)

  Description: This event occurs when task A has been deleted from job B.

  Action: If task A is in ReadyTasksData or AcquiredTasksData, remove it from that list.

- **TASK_READY**(Task A, Job B, Pool C)

  Description: This event occurs when task A is ready to be acquired by any user within pool C that was assigned to the role of the task.

  Action: If the pool is in UserPools, then add task A to ReadyTasksData. If task A is in AcquiredTasksData, remove it.

- **TASK_SET_ROLE**(Task A, Role B, Job C)

  Description: This event occurs when task A, within job C, has role B assigned.

  Action: If task A is in ReadyTasksData, get the pool that is assigned to role B. If the pool is in UserPools, then update the changed role and pool information for the task in ReadyTasksData. If the pool is not in UserPools, then remove the task A from ReadyTasksData, because the new role's pool does not contain CurrentUser.

  If task A is in AcquiredTasksData, get the pool that is assigned to role B and update the changed role and pool information for the task in AcquiredTasksData.

  If the task A is neither in ReadyTasksData or AcquiredTasksData, get the status of the task. If the status is ready, get the pool assigned to role B. If the pool is in UserPools, then add task A to ReadyTasksData.

- **TASK_TRANSFER**(Task A, User B, User C)

  Description: This event occurs when the task A is transferred from user B to user C. This means that user C has now acquired the task.

  Action: If CurrentUser is user B, remove task A from AcquiredTasksData. If CurrentUser is user C, add task A to AcquiredTasksData.

- **TASK_SKIP**(Task A, Job B, Pool C)

  Description: This event occurs when task A, which is part of job B, has been skipped.

  Action: If task A is in ReadyTasksData or AcquiredTasksData, remove it from that list.

- **TASK_WAITING**(Task A, Job B, Pool C)

  Description: This event occurs when task A has been put into a waiting status. A waiting task cannot be acquired.

Action: If task A is in ReadyTasksData or AcquiredTasksData, remove it from that list.

- **POOL_ADD_USER**(Pool A, User B)

  Description: This event occurs when user B has been added to pool A.

  Action: If the user B is CurrentUser, then add pool A to UserPools and refresh ReadyTasksData.

- **POOL_REMOVE_USER**(Pool A, User B)

  Description: This event occurs when user B has been removed from pool A.

  Action: If the user B is CurrentUser, then remove pool A from UserPools, and refresh ReadyTasksData.

- **ROLE_DEASSIGN**(Role A)

  Description: This event occurs when the pool has been de-assigned from role A. This generally means that users in the deassigned pool no longer have the ability to acquire and complete tasks that require role A.

  Action: Remove tasks that use role A within ReadyTasksData.

- **ROLE_ASSIGN_POOL**(Role A, Pool B)

  Description: This event occurs when pool B has been assigned to role A.

  Action: If there are any tasks in ReadyTasksData for role A, remove them if pool B is not in UserPools. Otherwise, if there are no such tasks and the pool B is in UserPools, refresh ReadyTasksData.

- **JOB_COMPLETE**(Job A)

  Description: This event occurs when job A has been completed (deleted) from the server. Tasks that make up job A are skipped if they are ready, or completed if they are acquired.

  Action: Remove every task from ReadyTasksData and AcquiredTasksData that has job A as its job.

There are more events that could be listened to by ActiveTaskList. The following events could be monitored to enhance the ActiveTaskList written for this thesis.

- **TASK_LATE_START**(Task A, Job B, Pool C)

  Description: This event occurs when task A, within job B, becomes ready past the due date set for the task.

  Action: In a more complicated ActiveTaskList, the application could change the color of the displayed task, alerting the user that it needs to be acquired and completed in a timely manner.

- **TASK_OVERDUE**(Task A, Job B, Pool C)

  Description: This event occurs when task A, within job B goes past the due date.

  Action: In a more complicated ActiveTaskList, you could change the color of the task to make it more apparent to users.

6.3 Notes on writing clients that use the Change Notification Facility

The following are some suggestions that make writing clients that use the change notification facility easier.

6.3.1 Plan out actions in response to events

When developing a client that listens for events, it is helpful, during the design, to list the event types the client is interested in. Given the list of events, plan what action the client will take for each event. While designing the ActiveTaskList, I found it helpful to make the list in section 6.2.2, detailing the event type, the data that is published with the event, and action that the client takes if it receives the event.

6.3.2 Efficiently determine interest in events

In addition to planning out the actions for each event type the client is interested in, it is important to efficiently determine whether the event directly affects the client. If care is not taken to efficiently determine event interest, the server will be bothered by queries that are not needed. The effects of inefficient planning are amplified when the client is

run on multiple hosts. During the design of ActiveTaskList, the actions were planned to make additional queries to the server only when necessary.

6.3.3 Altering client should display new state as user changes it

The purpose for developing InConcert clients is not to just view information on the InConcert server, but to also change it. When a client changes information on the server, there will be an event published that describes that change. For the purposes or this discussion, that particular client is called the altering client for the event that is published. If this client is designed correctly the altering client will take action on the published event, updating the graphical user interface to reflect the correct state of the server.

Does the altering client have to wait for the published event in order to update the graphical user interface? It can, but the behavior will look odd to the user running the client. The changes that the user makes won't appear on the screen until the event is published by the server, which could take as long as a minute given the default server configuration. If the event was published instantaneously, it would be more feasible to have the altering client wait. However, that is not the case, so it would be wise to update the graphical user interface and the state of the altering client.

If we alter the state of the altering client instantly to reflect the change that was made, sooner or later the published event reflecting that change will make it to the altering client. What does the altering client do with this event? It cannot distinguish the event it created from one that another client has made, so it performs the specified actions on the client's state. This situation can have undesirable effects unless the operations performed by the client in response to the published events are idempotent.

An operation is idempotent if repeated applications have the same effect as one. The set operations used in ActiveTaskList have this property. Adding the same element to the set once does exactly what is described, add the element to the set. If you perform the operation a hundred times in the row, adhering to the mathematical definition of set, there would be no effect on the set after the initial addition operation. Similarly, removing an element from a set has the same effect as removing it multiple times.

In the next chapter, we will discuss the performance of the ActiveTaskList in terms of the performance of the client and impact on the InConcert server. We will also discuss how the InConcert server and network in general are impacted by the addition of the change notification facility.

Chapter 7: Analysis of the Change Notification Facility

This chapter details the performance impact of providing and using the change notification facility. In considering the performance impact, we consider the extra work that both clients and the server have to perform, in addition to the extra network bandwidth that is used.

7.1 Client Performance

Clients that are using the change notification facility have to screen out events that do not interest them, and act on events that are of interest. This processing of events will cause a client to perform more work than its counterpart that does not use the change notification facility. This is basically the cost of listening. When writing clients that use this new facility, it is important to process the events in an efficient manner. The client should efficiently determine whether an event is of interest to them, as to not slow down the client with events that are of no interest.

With the current design, clients need to process every event that the server publishes. This is the largest weakness in the design of the change notification facility. Later in this chapter, we will discuss how the cost of listening could be decreased, so clients only receive events of types that interest them.

Although clients that use the change notification facility have to process every event published by the server, it is important to remember that using the facility adds new functionality that does not exist in previous clients. There are alternative designs for clients that can yield the same effect as using the change notification facility. For example, a client could poll the server periodically to update the information it is interested in. Later in this chapter, the analysis of the impact of ActiveTaskList discusses how the ActiveTaskList is better than a TaskList that polls periodically for information.

7.2 Server Performance

The additional work that the change notification facility adds to the InConcert server can be divided into two cases, the cost of publishing the events, and the cost of unnecessary queries made by clients in response to the published events.

7.2.1 The cost of publishing events

During the Event/Action daemon polling loop, the loop retrieves all of the new events, and publishes each one. Essentially, the Event/Action daemon now iterates through the set of new events twice, where before it iterated through them only once. The publishing could not be done at the same time as the trigger processing for the events because the new events that do not contain triggers are removed from the database table before the trigger processing loop. Therefore it was necessary to make a new query to the database to publish the events, before the new events table was altered. Considering that only one loop of activity and one database query was added to the Event/Action daemon, the publishing of the events by the server has a minimal effect on server performance.

7.2.2 The cost of unnecessary client queries

An effect of using the change notification facility is having clients make queries to the server in response to the published events. Some of these queries are useful, and lead to clients taking proper action. Other times, the client makes unnecessary queries, investigating events that end up not being related to the client. Although information can be added to events to provide a better context in order to determine client interest, clients can still make unnecessary queries. These unnecessary queries add additional load to the server.

7.3 Network Performance

How does the change notification facility affect the network? Additional network bandwidth is used by the server publishing events, and by the clients making additional queries in response to the published events. Since network bandwidth used by additional client queries is dependent on how the client is designed, it is possible that clients could be designed to never make additional queries on events. Therefore we will focus on the network bandwidth used by publishing events.

In considering the structure of how events are encoded, it is possible to get an upper bound on the size of a published event. In calculating the size, we have to consider the length of the field names, data, and the packaging provided by the Rendezvous messages. The maximum size encoded IcEvent includes all of the common data, and every optional data field being filled by its largest value. Luckily, the RvMsg class contains a method that reports its size in bytes [14]. Constructing the largest Rendezvous message that possibly encodes an IcEvent and using the size method supplied, we get a largest possible size of 2104 bytes. The smallest message size is 1009 bytes.

In order to determine the facility's effect on the network, we also need an idea of how fast the InConcert server publishes the events to the network. For purposes of getting this metric, I altered the published message to include the time that it was published. By producing a steady stream of activity on the server, it was possible to get a large amount of events queued up for publishing during the Event/Action polling loop. Using a client on the local network to listen for the events, it was possible to get the rate of events published per second. An InConcert server, running on a SPARC Ultra 10, can publish 1,000 events per second. This rate is dependent on the time it takes to encode an InConcert event as a Rendezvous message, and sending the message to the Rendezvous daemon that resides on the host that is running the InConcert server. The message is sent to the Rendezvous daemon via an Inter-Process Communication (IPC) mechanism that is implemented using a TCP connection to localhost (a host having a TCP connection to itself). It has been found that usage of a localhost TCP connection as an IPC mechanism is equal in speed and more robust than shared memory implementations of IPC [15].

Considering the maximum size of the message and the rate at which the server publishes messages, the InConcert server, under a constant stream of activity, could publish 2,104,000 bytes per second (2.104 megabytes per second) to the network. Note that this is using the worst case message size. Due to the action of the Event/Action daemon processing loop, the new events are published in periodic "bursts". At the beginning of the processing loop, the server will transmit 2.104 megabytes per second to the network,

until it runs out of events to process. The period of the bursts is dependent upon the polling period, which is set by the administrator of the server.

In practice, production installations of InConcert rarely process more than 180,000 tasks per hour (an example benchmark figure), typically much less. Since processing of a task involves three events (TASK_READY, TASK_ACQUIRE, TASK_COMPLETE) this gives an upper bound of 540,000 events per hour, or 150 events per second. At this rate of event generation, which is significantly less than the observed rate at which the Event/Action daemon can publish events, the network bandwidth utilization works out to 315,600 bytes per second in an absolute worst case, which appears quite manageable. The following table summarizes the change notification facility's impact on the network.

Minimum Rendezvous message size	1009 bytes
Maximum Rendezvous message size	2104 bytes
Maximum InConcert server event publishing rate	1000 events/sec
Maximum network bandwidth used by InConcert server publishing events	2.104 megabytes/sec
Maximum practical InConcert server event creation rate	150 events/sec
Maximum practical network bandwidth used by InConcert server publishing events	315,600 bytes/sec

Table 6: Summary of network impact of the change notification facility

7.4 Performance Impact of ActiveTaskList on the InConcert server

Other than the client's cost of listening, using the change notification facility can have an effect on the InConcert server. Here we will analyze how the ActiveTaskList impacts the InConcert server, and how it is better than another design that offers similar functionality.

When considering performance, it is important to note that different types of queries to the InConcert server have different performance costs. A query to find out which pool is assigned to a task (property query) is not as performance intensive as getting the list of ready or acquired tasks for a user (set query). In the ActiveTaskList, if we have to make a property query to find out which pool a task belongs to in order to prevent a set query, it is still better than blindly performing the set query.

7.4.1 ActiveTaskList compared to a polling TaskList

The ActiveTaskList provides the feature that the graphical user interface is updated when there is activity on the server that the user should know about. In this regard, it is better than the standard "static" TaskList. There are alternatives to the ActiveTaskList that would provide similar functionality. There could be a TaskList that polls every few minutes to refresh the ready tasks and acquired tasks. We will call this type of TaskList a polling TaskList.

The polling TaskList makes two set queries (for ready and acquired tasks) every period. Let's say that the polling TaskList refreshes every 15 minutes. During a 24 hour period of operation, the polling TaskList will make (4 polls/hour)*(2 set queries/poll)*(24 hours) = 192 set queries to the server. The polling TaskList will make these set queries regardless of whether information has changed on the server. The InConcert server gets bombarded with requests for information on a periodic basis from users on multiple hosts that are running polling TaskLists.

The polling TaskList is simpler to develop than the ActiveTaskList, because it only needs to make the queries in a timed manner. However, there is a benefit to the complexity of ActiveTaskList, because it will make set queries only when information pertaining to the user mandates it. It prevents bombarding the server with set queries periodically because it analyzes the information that it receives from the published events and makes a decision when to ask the server for information.

7.4.2 ActiveTaskList and unnecessary client queries

One case where the ActiveTaskList appears to be not as efficient as the polling TaskList is in the area of unnecessary client queries. ActiveTaskList performs extra property queries in order to determine client interest for only one event type, namely TASK_SET_ROLE. Upon receiving an event of this type, it tests to see whether or not the task associated with the event is in ReadyTasksData or AcquiredTasksData. If it is in either of those sets, it will make a property query to get the pool assigned to the newly set role. If the task is not in ReadyTasksData or AcquiredTasksData, the client makes a

property query to get the status of the task. If the status is ready, it makes another property query to get the pool of the task in order to determine if the pool of the newly assigned role is one the user is a member of. If the task is of no interest to the user running the client, the client will end up making one unnecessary client query (getting the status of the task) for every event of this type it receives.

The TASK_SET_ROLE event happens rarely when the task is ready or acquired, because setting the role of a task generally is performed during the design of a process template. If process templates are being designed at the same time ActiveTaskLists are running, then the ActiveTaskList will make the extra property query to determine whether the task is ready or acquired. During everyday use of the server, where the process templates are already designed and are used to create active processes, this event will rarely occur.

Another pitfall with the ActiveTaskList is when the application frequently receives events that cause it to make a set query to refresh the ready tasks. ActiveTaskList does this for 3 event types: POOL_ADD_USER, POOL_REMOVE_USER, and ROLE_ASSIGN_POOL. Although these circumstances are not strictly unnecessary queries, the set query needs to be made because the event indirectly denotes that information changed on the server that could be of interest to the user running the ActiveTaskList. The set query performed by the client could show that no new tasks are ready to be acquired.

Adding and removing users from pools occurs rarely during the everyday operation of the server, because pool assignments are generally done while setting up the server. However, assigning a pool to a role typically occurs during the startup of an active process, where every role for every task in the process is assigned a pool. Most of the tasks at this time are not ready, and are of no interest to any user running ActiveTaskList. This event occurs frequently during the everyday operation of the server, because active processes are created all the time.

The main reason ActiveTaskList listens to the ROLE_ASSIGN_POOL event is to refresh the list of ready tasks if a pool is assigned to the role as a replacement for another pool

and that role is used by a task that is ready. This situation rarely occurs. Given the described situations where this event occurs, it might be worth it to not listen for this event. However, not listening to this event would violate the consistency of ActiveTaskList covering every case upon which the graphical user interface should be updated.

From the above analysis, unfortunately there are cases where the ActiveTaskList can make property and set queries when it shouldn't. ActiveTaskList has the possibility to negatively impact server performance, but the impact is not as dramatic as initially anticipated.

7.5 Possible improvements of the Change Notification Facility

There are a few ways that the change notification facility could be improved to make the clients that listen to it more efficient. One improvement involves providing more information in the events that are published. The second improvement involves publishing the events to subjects that are partitioned according to event type.

7.5.1 Including more information in the published events

The InConcert events were intended for logging what has occurred in the InConcert server. Now they are being used to in a different manner, to inform clients of change. The change notification facility would be more effective if they included as much information possible in order to describe the context of the event. This would eliminate the possibility that the clients would have to do more investigation (making more property and set queries) in order to decide whether or not to take action on the event. This modification would involve altering the IcEvent data structure used by the clients and server, along with expanding the event database table to include more data fields.

7.5.2 Publishing events to subjects according to event type

According to our current design, when an IcEventService instance on the client receives any event, it delegates it to each of the IcEventListeners that have added themselves to the IcEventService. Each listener needs to filter the event upon receiving it from the

IcEventService. It might be better if the design of the IcEventService filtered out the event earlier on, so that each listener would receive only the events that it was interested in. This could be done if the IcEventService took advantage of the hierarchical naming of the subjects that TIB/Rendezvous could publish to.

Each event type would have its own subject under the "`eventservice`" hierarchy. For example, the TASK_READY event occurring on the server named "`finance`" would be published to the subject "`com.inconcert.finance.eventservice.TASK_READY`". On the client, the IcEventService would create a Rendezvous listener for each event type that its IcEventListeners wanted to listen to. This design would force the IcEventListeners to explicitly register the event types they were interested in, but the IcEventListeners would then never encounter event types that it had absolutely no interest in. This reduces the overall cost of listening discussed earlier. The next version of the change notification facility should include this design feature.

7.6 Ideas for more applications of the Change Notification Facility

The change notification facility created for this thesis can be used in many different contexts, not just for clients that listen for events and update their graphical user interface. Here are three different potential uses for the change notification facility.

7.6.1 Complete client cache maintenance

The events published by the InConcert server can be used by the client to update a cache of information that the client has received from the server. Depending on the type of event and the data that comes along with the event, the client will either update the cache with the data, or ask the server for more information. There are times when the information provided by the event is enough to adequately perform actions on the cache. There are instances, however, where the information provided by the event is not enough to adequately update the state of the cache to reflect the state that changed on the server. This may happen when a task dependency has been added, in which properties for the tasks surrounding the added dependency has changed. In this situation, the cache may

ask the server for a refresh of the complete dependency map. This idea has the potential to increase the performance of both the server and client.

7.6.2 A new message architecture for Agents

There are some InConcert applications that do not take input from a human user, or provide a graphical user interface, but wait to be prompted by the InConcert server in order to take action. These applications are known as agents. Before the development of the change notification facility, the server would contact the agents via a Remote Procedure Call (RPC). This RPC would be set up as a trigger upon the particular event that you wanted the agent to act on. The agents can now use the change notification facility to listen for the occurrence of a particular event, and perform its specified actions when the event is published. This method of contacting agents is more efficient due to the InConcert server publishing one event to contact potentially 100 agents, for example. With the RPC method of contacting agents, contacting 100 agents would require the server to perform 100 RPC calls. An additional benefit is the fact that listening to the change notification facility requires no setup on the server, while setting up an agent that listens for an RPC call requires an EventSpec, ActionSpec and a Trigger being set on the server [6].

7.6.3 Distribute the action processing of the Event/Action daemon

The change notification facility, used to its fullest extent, could also change the architecture of the InConcert server, allowing the Event/Action daemon to be broken up to different processes, which may be on different hosts. The portions of the daemon that perform actions based on the events and the triggers that are set could be broken up and put on other hosts. The Event daemon could publish the events in the manner implemented for this thesis, and there could be separate Action daemons that listen for these events and perform the actions based on the triggers. These Action daemons could be on separate hosts, and each daemon could handle a particular action, like sending e-mail, making an RPC call, or starting a job on the InConcert server. This change will increase the performance of the Event daemon polling loop, because it will not be

involved with performing the actions that correspond to the events, it will simply publish them to a subject listened to by the Action daemons.

Chapter 8: Conclusion

This thesis established that providing a change notification facility for the InConcert workflow management system is possible. The integration of the facility into the InConcert server required little additional design. The client interface involved minimal change to the existing InConcert Java API, and could be used to develop applications as complex as ActiveTaskList. In terms of a learning experience, this thesis has taught me a great deal about designing and writing software.

One of the most rewarding parts of my work was investigating the design of the InConcert server. The server has been refined over years by experienced professionals. I have never had the opportunity to look at that large of a product. The most exciting part of my work was creating and placing the server extension. I also got the opportunity to use TIB/Rendezvous, an industry standard messaging package.

Before discovering TIB/Rendezvous, I tried to implement my own reliable multicast messaging package. What resulted was a package that was limited in many ways. For example, the range of the publisher I developed was limited, only sending messages to the local network. Another limitation involved the size of the message, which could be as big as 540 bytes. Although none of the code written shows up in the final implementation of the solution for this thesis, I learned a great deal designing and writing it.

Vivek Ranadive, in his book The Power of Now, combines the publish/subscribe architecture with an event-driven management strategy to create a new paradigm for business: the event-driven corporation. This paradigm has the goal of focusing on and even anticipating customer needs by having employees take advantage of real-time, event-driven software to analyze information from various sources. Employees, behaving as knowledge workers, act near-instantly to the information that flows through their company. Companies can open up this flow of information to potential suppliers, who can, in turn, use the information to anticipate the company's needs and can compete with other suppliers for the company's business [11].

The work performed in this thesis helps bring the InConcert workflow management system closer to being a component in the event-driven enterprise described by Ranadive. This is an important direction to take for the InConcert server, because the process organization capability of InConcert can be the brains of an event driven environment. InConcert can be used to sense the real-time information flowing throughout a corporation and use it to set up higher level goals, in the form of InConcert processes. Additionally, this thesis shows that the clients and agents in the event-driven enterprise can use the InConcert server in a new way. Clients and agents can now respond to the progress that is made within InConcert processes, forming a more complex enterprise-wide nervous system.

Bibliography:

1. Banavar, G. et al., An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems, IBM Research Division (Yorktown Heights, NY), Jan 22, 1999.

2. DellaFera, C.A. and Eichin, M.W., The Zephyr Notification Service, Proceedings of the USENIX Winter Conference, Dallas TX: USENIX Association, 1988.

3. Floyd, S. et al, A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, ACM SIGCOMM '95, August 7, 1995.

4. Gamma, E. et al., Design Patterns, Elements of Reuseable Object-Oriented Software, Addison-Wesley (Reading, MA), 1995.

5. TIBCO Software Inc., TIB/InConcert UNIX Installation and Administrator's Guide, TIBCO Software Inc. (Cambridge, MA), 2000.

6. TIBCO Software Inc., TIB/InConcert C Programmer's Guide, TIBCO Software Inc. (Cambridge, MA), 2000.

7. Lin, J., Paul, S., RMTP: A Reliable Multicast Transport Protocol, Proceedings of IEEE INFOCOM '96, March 1996, 1414-1424.

8. Oki, B., Pfuegl, M., Siegel, A., Skeen, D., The Information Bus – An Architecture for Extensible Distributed Systems, SIGOPS, ACM Press, 1993, 58-68.

9. Patterson J., Day M., Kucan J., Notification Servers for Synchronous Groupware, Computer Supported Cooperative Work '96, (Cambridge, MA), 122-129, 1996.

10. Ramduny, D., Dix, A. and Rodden, T., Exploring the design space for notification servers, Computer Supported Cooperative Work '98, (Seattle, WA), 227-235, 1998.

11. Ranadive, V., The Power of Now, How Winning Companies Sense and Respond to Change Using Real-Time Technology, McGraw-Hill, (New York, NY), 1999.

12. Sweeny, T., Unreal Networking Architecture, http://unreal.epicgames.com/Network.htm, 1999.

13. TIBCO Software Inc., TIB/Rendezvous Concepts Guide Release 5.0, TIBCO Software Inc., (Palo Alto, CA), 1998.

14. TIBCO Software Inc., TIB/Rendezvous C++ Guide Release 5.0, TIBCO Software Inc., (Palo Alto, CA), 1998.

15. TIBCO Software Inc., TIB/Rendezvous 5.0 FAQ, TIBCO Software Inc., http://www.rv.tibco.com/faq.html, 1998.

16. TIBCO Software Inc., TIB/Rendezvous Java Guide Release 5.0, TIBCO Software Inc., (Palo Alto, CA), 1998.