An Implementation of the OASIS Business Transaction Protocol on the CORBA Activity Service

By Adam D. Barker

MSc SDIA, September 2002

Supervisor: Dr. Mark Little

CONTENTS

CONTENTS..3

TABLE OF FIGURES..6

ACKNOLEDGMENTS..8

ABSTRACT..9

CHAPTER ONE - INTRODUCTION..10

CHAPTER TWO - TRANSACTIONS..10

2.1 - What is a Transaction?..10

2.2 – Traditional Atomic Transactions..11

2.3 – Distributed Transactions and the Commit Protocol..11

2.3.1 – Two Phase Commit..11

2.4 – Are ACID Transactions Sufficient?..13

2.4.1 - Extended Transactions..13

2.5 – Business Transaction Protocol..14

CHAPTER THREE – THE COORDINATION MODELS..16

3.1 – The Activity Service Framework..16

3.1.1 – How the components of the Activity Service interact..17

3.2 – A Closer look at The Business Transaction Protocol..21

3.2.1 – Superior..21

3.2.1.1 – BTP Atoms..21

3.2.1.2 – BTP Cohesions..22

3.2.2 – Inferior..22

3.2.3 - The Superior:Inferior relationship..22

3.2.4 - Services..23

3.2.4 – Participant..23

3.2.5 – Decider..23

3.2.6 – Terminator..23
3.2.7 – BTP exposes both phases to the user..24
3.2.8 – Opinion from every direction..24
3.3 – Real World Example using BTP...25
3.3.1 - Multiple Party Atomic Transaction...25
3.3.2 - Cohesion Example..26
CHAPTER FOUR – BTP ON THE ACTIVITY SERVICE...........................27
4.1 – Interfaces to BTP...27
4.1.1 - Inferior Interface...27
4.1.2 - Superior Interface..28
4.1.3 - Atom Interface..28
4.1.4 - Cohesion Interface...28
4.1.4.1 – Prepare Inferiors...28
4.1.4.2 – Confirm Transaction..29
4.1.4.3 – Cancel Transaction..30
4.1.4.4 – Cancel Inferiors..30
4.2 – Architecture of the Solution...31
4.2.1 – Overall View..31
4.2.2 - How do Superiors and Inferiors talk to each other?...................................31
4.2.2.1 – Enrolling an Inferior with an Atom...31
4.2.2.2 – Enrolling an Atom with a Cohesion..32
4.2.3 - Asynchronous Behaviour...34
4.2.3.1 – Inferiors can Prepare early..34
4.2.3.2 – Inferiors can Confirm or Cancel early...34
4.2.4 – Signal Sets...36
4.2.4.1 - Atom Signal Sets..36
4.2.4.2 - Cohesion Signal Sets..37

4.2.4.3 – Signal Sets can only be used once..38
4.2.5 - How Errors are handled using the Activity Service..........................38
4.3 - Example using the BTP API..39
4.3.1 – The back end...39
4.3.2 – The users view...41
CHAPTER FIVE – CONCLUDING REMARKS..44
5.1 - Comments..44
5.1.1 –The Activity Service, a useful tool?..44
5.1.2 – BTP, the way of the future?..45
5.2 – Future Work..45
REFERENCES 47
URL's 47

TABLE OF FIGURES

Figure 1: Illustrations of Two Phase commit with two participants....12

Figure 2: Global decision applied to participants....12

Figure 3: An example of a logical long running transaction....13

Figure 4: A transaction can continue to make forward progress....14

Figure 5: Actions register interest with the Signal Set....17

Figure 6: Coordinator requests a Signal from the Signal Set....18

Figure 7: Coordinator forwards the Signal to the Actions....18

Figure 8: Actions return Outcomes....18

Figure 9: 2PC modelled on the Activity service....19

Figure 10: An overview of the key roles in BTP....21

Figure 11: A Superior with two enrolled Inferiors....22

Figure 12: Superior:Inferior relationships can be nested....22

Figure 13: BTP interactions....23

Figure 14: Both phases of the Commit Protocol....24

Figure 15: Confirm Transaction scenarios....29

Figure 16: Confirm Transaction with an Inferiors list....30

Figure 17: Components of the implementation....31

Figure 18: Creation of an Atom....31

Figure 19: Enrolling an Inferior with an Atom....32

Figure 20: How Signals are transmitted....32

Figure 21: Enrolling an Atom with a Cohesion....33

Figure 22: Prepare Inferiors sequence diagram....33

Figure 23: Prepare can be issued early....34

Figure 24: Asynchronous communication....35

Figure 25: Signal Sets to implement BTP....36

Figure 26: StageOne Signal Set 36

Figure 27: StageTwo Signal Set 36

Figure 28: Prepare Inferiors 37

Figure 29: Confirm Transaction 37

Figure 30: Cancel Inferiors 37

Figure 31: How errors are detected and thrown as Exceptions 38

Figure 32: Night out or night in that is the question! 41

ABSTRACT

Properties of traditional transactions are described by the ACID (Atomicity, Consistency, Isolation, Durability) acronym. These transactions are typically short lived and are well suited in tightly coupled homogenous environments, where resources are owned by the same organisation and are designed to work together as a unit. However ACID transactions are not appropriate for everything, especially 'business to business interactions', which are designed to run over long periods of time. The CORBA Activity Service was developed as a generic framework for coordinating units of computation, it was developed to be a flexible, reusable tool to support the implementation of extended transaction models.

BTP is a specific extended transaction model that allows coordination of resources which are exposed by multiple autonomous organizations. This model relaxes the traditional ACID properties and forms a protocol that can run for long periods of time over the inherently unreliable environment that is the Internet. This project aims to demonstrate if the CORBA Activity Service is a sufficiently flexible model to provide an implementation of BTP and whether the functionality provided by the framework is enough to support the complex interactions specified by the protocol.

CHAPTER ONE

INTRODUCTION

Properties of traditional transactions are described by the ACID (Atomicity, Consistency, Integrity, Durability) acronym. They are an important tool in the development of enterprise business applications; they enable fair exchange in atomic 'all or nothing' behaviour. These semantics work well in a tightly coupled homogenous environment, where they are typically short lived, message delivery can be guaranteed and often one company owns all the resources that are being accessed. However business-to-business (B2B) interactions are becoming more commonplace, these interactions can run for minutes, hours or even days and can be incredibly complex. The traditional semantics of ACID transactions are not appropriate to be applied to these long running B2B interactions. A long running ACID transaction would mean that the resources being accessed would be exclusively locked for the duration of the transaction, reducing the concurrency in the system to an unacceptable level. Also if the transaction had to be undone for any reason much valuable work would have to be rolled back. One way to eradicate these problems is to use an extended transaction. An extended transaction is a way of structuring a long running transaction into many short-lived ACID transactions. The CORBA Activity Service is a generic framework that is focused on the coordination of units of computation; it was developed to be a flexible, reusable tool to support the implementation of extended transaction models.

This project is focused on providing a prototype implementation of the OASIS Business Transaction Protocol (BTP) using the functionality of the Activity Service framework. BTP is a standardized extended transaction model. It is focused on coordinating resources in the loosely coupled business-to-business (B2B) space, where the resources taking part in the transaction are not exclusively owned by one business but several businesses. This model relaxes some of the ACID properties and allows a more flexible protocol that can be run over the loosely coupled, unreliable environment, which is the Internet. The main aim of this work is to see if the Activity Service framework is a sufficient coordination model to support the complex interactions of BTP.

CHAPTER TWO

TRANSACTIONS

This chapter introduces the concept of a transaction, looks at the traditional model of ACID semantics and how these can be loosened with extended transactions and the Business Transaction Protocol.

2.1 - What is a Transaction?

"A Transaction is an interaction in the real world, usually between an enterprise and a person, where something is exchanged. It could involve exchanging money, products, information, requests for services and so on. " [Bernstein97]

A typical example of a transaction would be the purchasing of your shopping at a supermarket, the exchange of food items for money. This needs to be a fair exchange, neither party should loose out, the customer gets the food and the shop gets the money in exchange.

"An on-line transaction is the execution of a program that performs an administrative function by accessing a shared database, usually on behalf of an online user" When the word transaction is referred to from now on it relates to the execution of a program which contains the steps involved in a business transaction, for example recording the sale of a book from a book seller and debiting the inventory.

2.2 – Traditional Atomic Transactions

Atomic transactions are used for controlling operations on persistent shared information; this data can be files, objects, databases records etc. Atomic Transactions provide a simple model of success or failure. A transaction either commits (all its actions happen), or it aborts (all its actions are undone). This all-or-nothing quality is used to ensure consistent state changes.

A typical example of an Atomic Transaction is the situation where a customer wants to withdraw some money from his account using an ATM. This operation needs to be executed in the scope of a transaction, because if something goes wrong (the machine jams or there is not enough money in the ATM) then the money should not be debited from the customers account. It requires all or nothing behaviour, the user gets his money and the money is debited from the customers account or nothing happens, and the state

remains unchanged. Hence if something was to go wrong during the processing of the transaction it needs to look 'logically' as if nothing ever happened.

Transactions have what is known as ACID properties, this acronym describes the classical all or nothing behaviour that is associated with transactions:

§ **Atomicity:** A transaction needs to be atomic (all or nothing), this means that it executes completely or not at all. If one part of the transaction fails no part of the transaction program can be executed. If for any reason the transaction cannot be completed, everything this transaction changed can be restored to the state it was in prior to the start of the transaction, via a rollback operation.

§ **Consistency:** Shared resources of the transaction should remain consistent. A transaction must take the system from one consistent state to another. Consistent state meaning that certain conditions are met defined by the business, e.g. ensuring an overdraft status is less than the total overdraft that an account will grant.

§ **Isolation:** Each transaction accesses resources as if there were no other concurrent transactions; the transaction executes without any interference from other transactions and has exclusive access to the resource(s). It's as if the system running the transactions is doing so in a serial order, one after the other. Modifications of resources by the current running transaction are not visible to other transactions until it is finished.

§ **Durability:** Once the transaction has finished, all state changes made to objects and data are saved on permanent storage.

2.3 – Distributed Transactions and the Commit Protocol

When a transaction is executed in a local or distributed environment the ACID properties must be maintained. This is more difficult to achieve if the transaction resources are physically located on different machines. A way is needed to ensure that if the transaction decides to commit, the updates are done on all systems.

The main problem with executing transactions in a distributed environment is that any of the nodes could arbitrarily crash, and because the data (objects, files etc) are physically located on different machines it is more difficult to detect a node crash. For

example if the transaction commits the updates on one system but a second system fails (crashes) before the transaction commits then this could lead to an inconsistent state. This would result in part of the work involved in the transaction being completed and part not; this violates the Atomic property we saw before.

The solution to this problem is to use a commit protocol; the most common commit protocol used in transaction systems is called 'two phase commit', but other forms of commit protocol exist such as 'three phase commit' if more reliability is needed.

2.3.1 – Two Phase Commit

The Two-Phase Commit (2PC) protocol is a simple protocol to achieve consensus between all parties taking part in the transaction before allowing the transaction to terminate in a consistent state.

There are two types of processes involved in this protocol that need to be understood; a ***Coordinator*** that decides whether to reach a global commit or abort decision, issues messages and receives the reply from the ***participants***. The ***participants*** effectively sit in front of the resources (Data – Base, file etc) that the transaction is trying to update. It's the participants that vote whether to commit or abort after receiving messages from the coordinator and it's the participants that access the resources and finally update the resources if the transaction decides to commit.

The commit decision is made according to the ***global commit rule***:

- If even one participant votes to abort the transaction, the coordinator has to reach a ***global abort decision***.
- If all participants vote to commit the transaction, the coordinator has to reach a ***global commit decision***.

2PC as its name suggests consists of two phases, this is a description of a typical way in which the protocol may execute, other variations which include some optimizations exist:

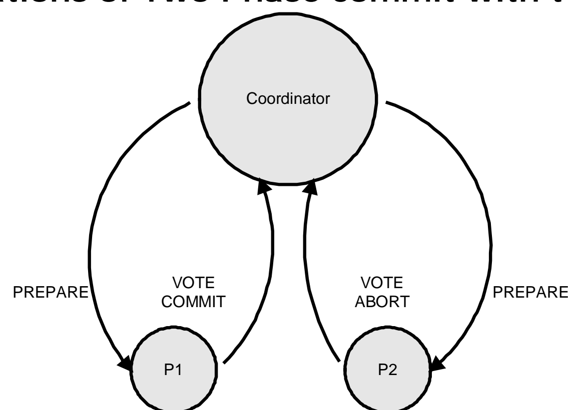
Phase 1: A coordinator process is started (usually at the site where the transaction is initialized), writes a begin record in its log, sends a ***prepare*** message to the participants, and enters the wait state. This message may also contain a unique transaction id (TID), which all-further messages in this protocol run.

When a participant receives a ***prepare*** message, it checks if it can commit to the transaction. If it can, the participant writes a ready record in its log, sends a v***ote_commit*** message to the coordinator, and enters the ready state. Otherwise, the participant decides to unilaterally abort the transaction, it writes an abort record in the log and sends a ***vote_abort*** message to the coordinator. It enters the abort state and can forget about the transaction.

Phase 2: After the coordinator has received votes from all participants it decides whether to commit or abort the transaction according to the global commit rule, and writes this decision in the log. If the decision is to commit, it sends a ***global_commit*** message to all participants. Otherwise, it sends a ***global_abort*** message to all participants that voted to commit. Finally, it writes an end of transaction record in its log. The participants finish the transaction according to the decision and write the result in their logs.

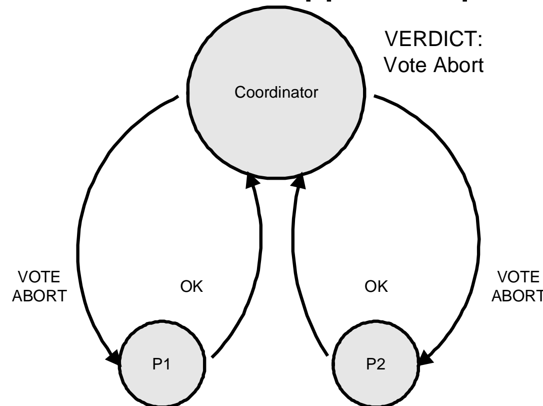
Shown in the two diagrams below is an example of how two-phase commit might execute with two participants. The coordinator has issued both participants with the prepare message. P1 has replied Vote Commit where as P2 has replied with Vote Abort.

Figure 1: Illustrations of Two Phase commit with two participants



Due to the global abort rule, all of the participants must reply vote commit to the prepare message to reach a global commit decision; hence the Coordinator decides to cancel the transaction, as P2 replied vote abort to the prepare message. Vote_abort is issued to the participants, which undo any state changes.

Figure 2: Global decision applied to participants



The Commit protocol ensures that the ACID properties can be maintained over a distributed environment, enabling the transaction to terminate in a consistent state even if a node crashes that is physically remote from the coordinator.

2.4 – Are ACID Transactions Sufficient?

Distributed Objects and ACID transactions provide a way of coordinating resources to ensure that state changes are consistent. This is still the most common form of transaction used. However ACID transactions are normally short-lived, possibly 1000's of transactions being executed in a second. ACID transactions are not so well suited to long-lived, otherwise known as extended transactions, which could run for minutes, hours or even days.

There are several problems with having long lived traditional ACID transactions:

§ A long running ACID transaction reduces the concurrency in the system to an unacceptable level, by locking resources for the entire duration of the transaction, from when the transaction is begun until when it is aborted or committed. This means that no other transactions can have access to the resource. If the transaction is long lived this is a very inefficient way of exposing a resource. If the resource happened to be a book from an online bookseller and the transaction aborts then no one else could have purchased that book during the time of the long-lived transaction, potentially loosing out on a sale.

§ If the long-lived ACID transaction aborts then much valuable work will have to be undone, either resulting in a compensation

transaction to undo the work or another attempt to complete the work being made.

§ ACID transactions are an atomic unit, all or nothing behaviour; this is not always necessary in some applications, where some kind of logical choice may be needed.

If a transaction is long lived then often these strict ACID properties can be relaxed a little, hence the need for non - ACID or Extended Transactions. It's worth mentioning a transaction can itself have transactions nested inside it, much like a program has sub routines embedded within it. For example a transaction to pay a bill could have two nested transactions, one for debiting the cardholders account and one for crediting the companies account. The main transaction is known as a top-level transaction. For more information on nested transactions refer to [Bernstein97], chapter eleven.

2.4.1 - Extended Transactions

An Extended transaction has relaxed ACID properties, and may be used when the transaction is going to be long running. An extended transaction can be structured as many short duration top-level (not executing in the scope of another transaction) transactions, which form a 'logical' extended transaction. Therefore any resource(s) that are accessed by the short-lived top-level transaction are only locked until it terminates, not for the entire length of the transaction. The notion of an extended transaction eradicates the three problems mentioned above. The resources are not held onto throughout the entire length of the long running transaction, therefore other transactions can access these shared resources more quickly as they are not locked for as long. This improves the concurrency and efficiency in the system.

A long running or extended transaction could be made up of a series of short-lived top-level transactions, as illustrated by the diagram below. The outer dotted line represents the boundaries of the 'logical' extended transaction and the solid circles represent short duration top-level transactions.

Figure 3: An example of a logical long running transaction

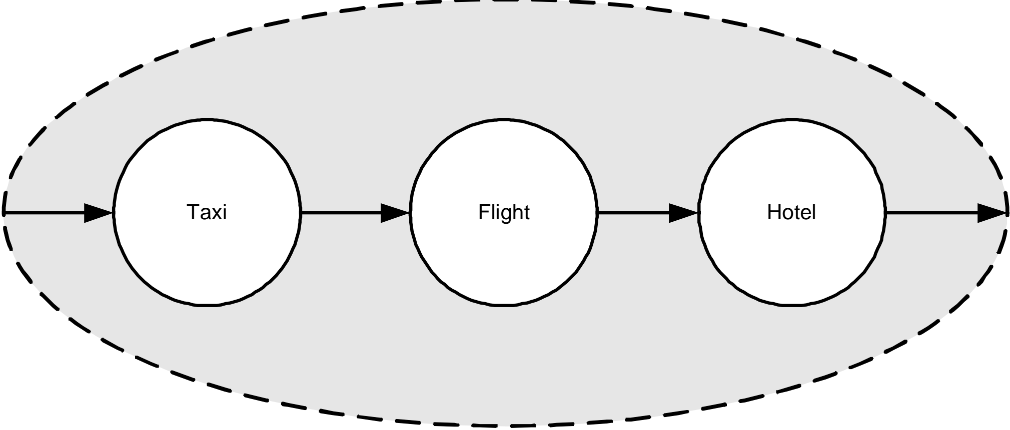
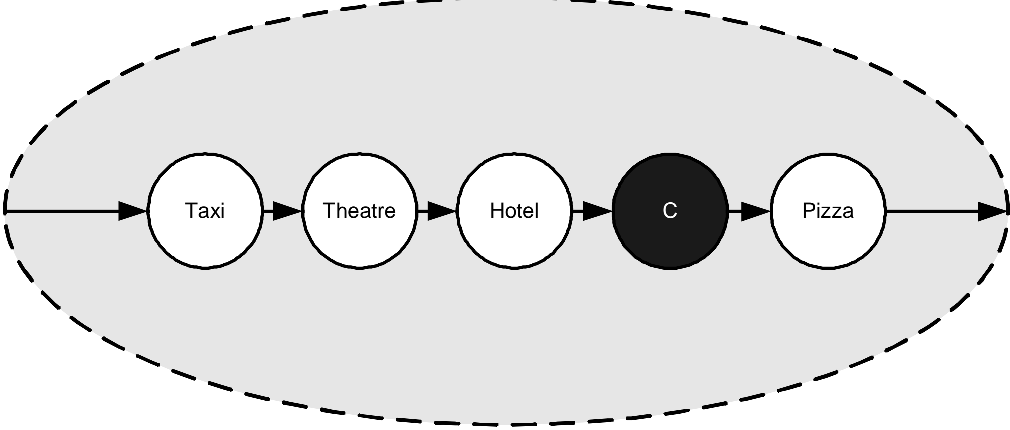


Figure 3 shows the booking of a holiday. A user wants to book a taxi to the airport, a flight to the holiday destination and a Hotel to stay in, in one atomic unit. The user either wants all of the above to happen or none of it too. These three short top-level transactions form a 'logical' extended transaction. If the above example were structured as a long running ACID transaction then even though the taxi work may have been completed the resources (e.g. Taxi database) would only be released when all of the work of the transaction has completed. With the extended transaction model the resources are only locked for as long as the individual top-level transaction (Taxi, Flight, Hotel) takes.

An extended transaction may still continue to make forward progress if one of the short-lived transactions fails. Take the scenario where a user wants to organise a night out, this involves booking a taxi to the theatre, booking the seat in the theatre and booking somewhere to sleep in a hotel. The user wants all of this work to be done as a single atomic unit, either all of it is booked or none of it is booked. This is illustrated in Figure 4.

Figure 4: A transaction can continue to make forward progress



In the example illustrated by Figure 4, booking the room in a hotel is not possible because the hotel is fully booked, hence all of the work cannot be completed. However the transaction can still make forward progress. If the user can't organise this night out he decides to stay in and order a Pizza instead. He can only do this if all the work is undone, this means rolling back the taxi and the theatre by invoking a compensation transaction (illustrated by C on the diagram). This compensation transaction undoes all the work that the previous transactions (Taxi and Theatre) completed. If this is successful and the Taxi and the Theatre can be rolled back then the Pizza can be booked instead.

There are many extended transaction models, designed for different purposes and are often application specific. This project is concerned with the OASIS Business Transaction Protocol.

2.5 – Business Transaction Protocol

Traditional Atomic transactions work well in a tightly coupled homogenous environment, for example an ATM machine, where the same company owns the databases, message delivery can be guaranteed and retransmission is possible. However these tightly coupled Atomic semantics do not fit in well with the architecture of the Internet, where message delivery cannot be guaranteed and transactions may be long running. More and more businesses are trying to integrate there existing architectures with the heterogeneous architecture of the Internet, therefore a standard coordination protocol is needed to tie all the ends up between businesses where one party does not control all of the resources that need to be accessed in a transaction.

The OASIS Business Transaction protocol (BTP) is a standardized extended transaction protocol designed for Business-to-Business communication using long running transactions, it was developed by BEA, Hewlett-Packard, Sun Microsystems and Oracle. These transactions could run in the order of hours, days etc. ACID transactions of this length would simply not be feasible so a new model, BTP has been developed to enable coordination of resources from heterogeneous environments, eradicating the exclusive locking of a resource by a transaction.

For example, the process of an online bookshop may well reserve books for an individual for a specific period of time, but if the individual does not purchase the books within that time period then they will be 'put back on the shelf' for others to purchase. If this situation were modelled using traditional Atomic Transactions then the book would

be locked until the transaction had terminated (committed or aborted), meaning that no other transaction could access the resource. Businesses cannot afford to lock the resources for the duration of the transaction as they do in ACID transactions.

BTP is designed to allow application work to be coordinated between multiple participants owned or controlled by autonomous organisations. BTP uses a two-phase coordination protocol to ensure the outcome of the transaction is a consistent result. It is designed for transactions in a 'loosely coupled' environment where failures and message delivery cannot be guaranteed. Business-to-Business interactions can be very complex, involving many parties spanning many different organizations hence the need for a specific coordination model that can span multiple autonomous organisations.

BTP is an interoperation protocol, it simply defines the message set and the expected behaviour of the senders and receivers, no implementation architecture is specified. It has two types of transactional behaviour, an atom and a cohesion, which are more suited to the architecture of the Internet which is inherently unreliable. In short what BTP provides is transaction abstractions suited to B2B interactions, with loose coupling and no requirements on what happens on the back end systems that expose a resource.

BTP is a generic specification; it can be implemented with any communications medium as the underlying carrying mechanism. For the purpose of this project it will be implemented upon the services offered by the Activity Service, which uses CORBA as the communications medium. However BTP is ideally suited to the emerging world of Web Services, where messages are schematized in XML and carried using SOAP. It's ideally intended for the world of web services for a number of reasons. Firstly it's executing over the Internet, which is a loosely coupled environment, message delivery cannot be guaranteed, as many companies own the infrastructure. Its an untrustworthy environment, the resources and participants are owned by multiple companies some of which you may not fully trust, therefore a standardised protocol is needed to ensure fair exchange between all participants in the transaction.

CHAPTER THREE

THE COORDINATION MODELS

This chapter looks in detail at the Activity Service framework, the components that make up its coordination model and how these interact. It also addresses the Business Transaction Protocol in more detail and puts the concepts into context with a real life example.

3.1 – The Activity Service Framework

The Activity Service is a low level generic framework that provides users with a general event signalling mechanism to allow the coordination of Activities (units of application specific computation) in a manner specified by the model under consideration. For the purposes of this project the model will be BTP. Various coordination models, including extended transactions can be placed on top of this framework to enable the programmer to concentrate on actually developing the model not the application specific mechanisms for coordination. This means a highly flexible, reusable framework which reduces the work that has to be carried out by a developer. The Activity Service is concerned with the control and coordination of Activities, leaving the semantics of these Activities to the application programmer. The Activity Service does not specify the implementation details of how the Activities should be coordinated, only providing interfaces for coordination to occur. To understand how the implementation of the project has been fitted together it is necessary to take a close look at the Activity Service and BTP breaking both down into their constituent components. The architecture of the solution can then be examined in more detail.

The Activity Service framework has the following core components that form the basis of the coordination model:

§ **Activity:** An Activity is a unit of Distributed Work that may or may not be transactional. An Activity is created, made to run and then completed. They are designed to run over long periods of time, can be nested and communicate to other activities. When an Activity is completed it returns its final outcome. This outcome can contain application specific data, and for example whether or not the units of

computation have been successful. This can then in turn determine the subsequent flow of control to other Activities.

§ **Action:** Activities interact with one another through Signals and Actions. An Action is a smaller unit of work inside an Activity (e.g. in the example shown in Section 2.4.1, the Actions could be book Taxi, book Theatre and book Hotel); An Activity can contain multiple Actions. When an Activity requires coordination, actions will be invoked with the specified Signal. An Action may then use the information encoded in the signal in an application specific manner. When the operation has been performed with the Signal an outcome is returned to the Activity Coordinator. This outcome is then passed back to the Signal Set, which contains the logic to decide which signal (if any) to return next when the Coordinator asks for the next signal.

§ **Signal:** A signal is simply a way to encode information, which is passed to Actions via the Activity Coordinator. It has a signal name, is associated with a Signal Set and can, if required contain application specific data.

§ **Signal Set:** Signals are associated with Signal Sets. A Signal Set represents the set of signals that are required to achieve some goal (e.g. Two Phase Commit or in the projects' case BTP). Actions register interest with particular Signal Sets; multiple Signal Sets may be registered with a Coordinator.

The logic of which signal to send to an action is encoded within the Signal Set. The Signal Set is application specific; different Signal Sets will be used for different goals.

When all signals have been sent, the Activities final outcome can be obtained from the Signal Set, the final outcome often contains whether or not the Activity's' work has been successful.

§ **Activity Coordinator:** The coordinator is responsible for coordinating the interactions between Activities through signals and Actions. One Activity one Activity coordinator. This is a generic

object and is relatively simple, it knows nothing of the application logic than underpins the coordination model, therefore the Coordinator never changes, it is the same for each use of the Activity Service.

The Activity Service architecture is highly flexible and contains reusable, generic components that don't need to change. The application specific work is contained within the Signal Sets and Actions and how these components interact.

The communication protocol that is used between different components of the Activity Service is CORBA, and the following IDL interfaces are provided for the components that need to be defined by the user:

An Action has the following defined interface:

```
interface Action
{
	Outcome process_signal(in Signal sig) raises ActionError;
};
```

A Signal is simply a struct that can contain application specific data:

```
struct Signal
{
	string signal_name;
	string signal_set_name;
	any application_specific_data;
};
```

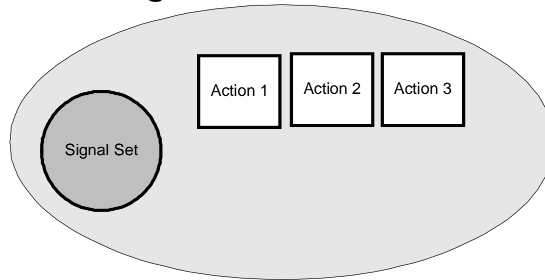
Finally a Signal Set has the interface defined below:

```
interface SignalSet
{
	readonly attribute string signal_set_name;
	Signal get_outcome () raises(SignalSetActive);
	Boolean set_response (in Outcome response, out Boolean nextSignal)
	                              Raises (SignalSetActive);
	Void set_completion_status(in CompletionStatus cs);
	CompletionStatus get_completion_status();
};
```

3.1.1 – How the components of the Activity Service interact

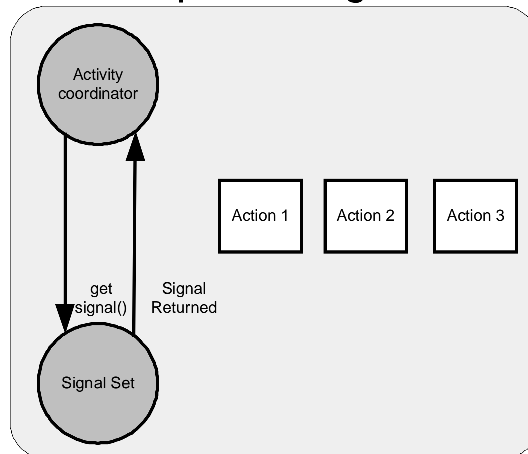
Before any interactions between the components can take place, the user needs to register the Signal Sets with the Activity Coordinator and then register Actions with the Signal Sets(s). This means that once the Signal Set is processed, actions that are registered to it are passed the relevant signal and the coordination can begin. Action 1, 2 and 3 and registered with the Signal Set as demonstrated below:

Figure 5: Actions register interest with the Signal Set



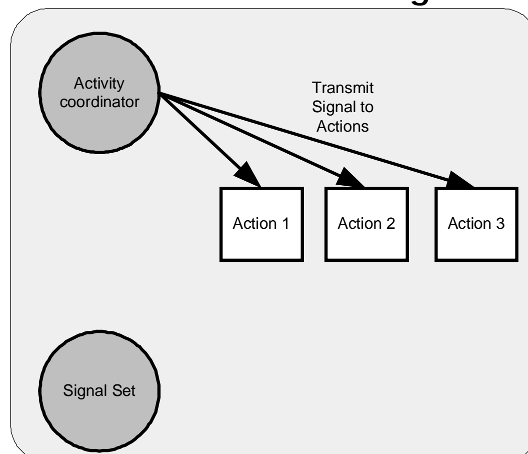
A basic overview of the Activity Service is shown in the figure below; the diagram illustrates the interactions between the Activity Coordinator, the Signal Set and the registered Actions. Actions have to register interest with Signal Sets. When the Activity begins to run the Coordinator goes to the Signal Set to get a signal, it does this by calling the ***get_signal()*** method.

Figure 6: Coordinator requests a Signal from the Signal Set



A signal is returned from the Signal Set which is then sent to each registered action in turn, by calling the ***process_signal()*** method on each Action which is registered with that Signal Set, passing the signal as a parameter in the method.

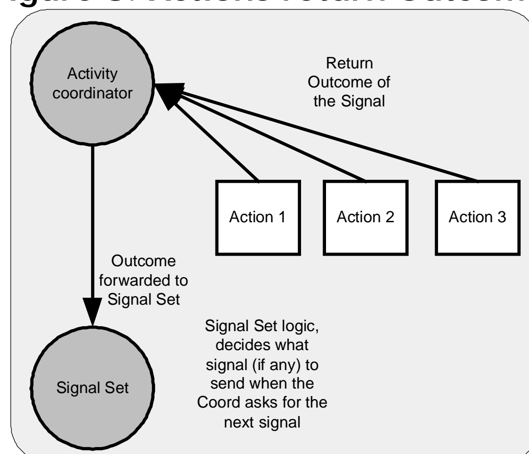
Figure 7: Coordinator forwards the Signal to the Actions



When the Action receives the signal it will return an Outcome to the Coordinator as a result of the ***process_signal()*** method. An outcome is simply a way of returning a response; this outcome can have application specific data embedded within it that the Signal Set will understand.

This outcome is forwarded to the Signal Set, using the ***Set_Response()*** method on the Signal Set, passing the outcome as a parameter. After receiving the outcome the Signal Set can return two possible variables. The first one is a Boolean which states whether or not the Action wishes to receive any more Signals from the current Signal Set, this will be true if the Action does require more signals. The second possible parameter is the ***next_signal*** variable, if this is set to true then the Coordinator abandons sending the current signal to any more actions and goes back early to the Signal Set for the next signal.

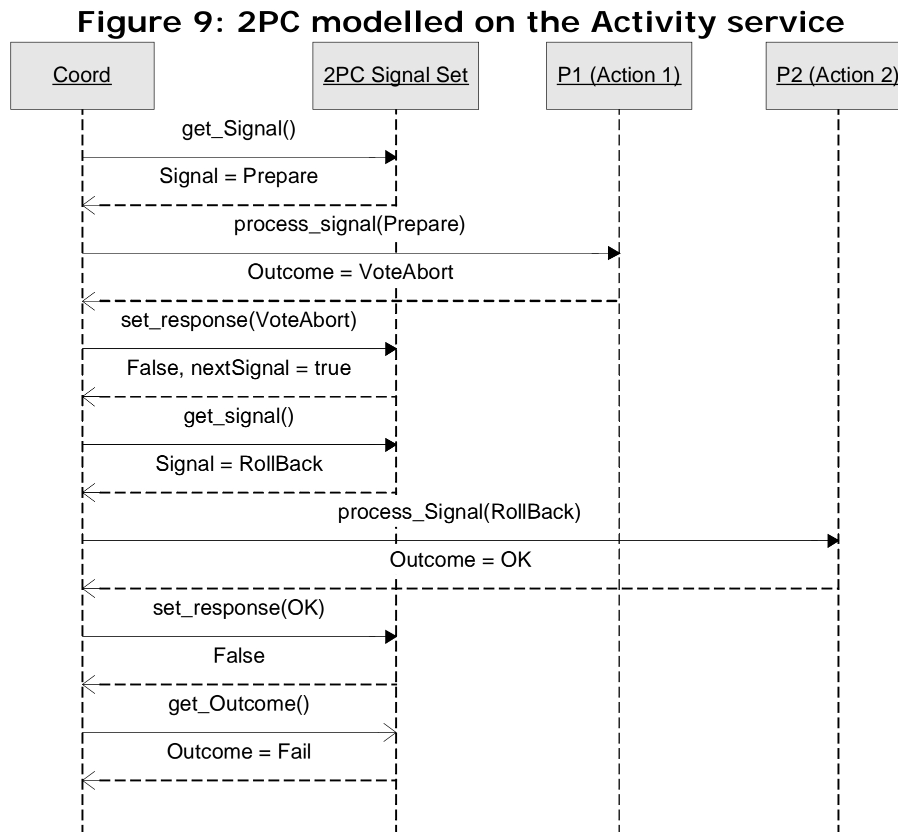
Figure 8: Actions return Outcomes



A Signal Set is application specific and different Signal Sets will be developed for different coordination models. When all Signals have been sent a final outcome will be returned.

The example illustrated below and explained by the text underneath demonstrates how two phase commit can be implemented on the Activity Service and how failures are handled:

Figure 9: 2PC modelled on the Activity service



The figure illustrated above represents how two phase commit would be modelled on top of the Activity Service, in this example the transaction contains two participants (Actions) and one of them (P1) decides to Vote Abort to the prepare stage. In reference to the figure shown above:

- The AC calls ***get_signal()*** on the 2PC Signal Set. The signal returned is "Prepare", the "Prepare" signal is sent P1 (the registered Action). P1 replies with the Outcome "VoteAbort". The "VoteAbort" outcome means that the transaction cannot continue because if one participant aborts (ops out) of the transaction then it cannot continue (global abort rule).
- The AC now calls ***set_response(VoteAbort)*** on the 2PC Signal Set. This method is called to notify the Signal Set of the response from P1. The Boolean returned from the set_response() method would be False, meaning that P1 is not interested in any more signals from the 2PC Signal Set. The ***next_signal*** variable would be set to 'true' because the prepare signal is no longer valid to any

registered actions, in this case P2. This means that the AC will go back earlier than it would normally to the 2PC Signal Set, getting the next signal.

§ When the AC calls ***next_signal()*** on the 2PC Signal Set, this time the "Roll-Back" signal will be returned. Rollback needs to be sent to P2 to tell it that the transaction is not going ahead and to undo any work that it has started.

§ The outcome is returned, in this case "OK" and is used in the ***set_response()*** method on the 2PC Signal Set. The Boolean value returned from this method would be false, indicating that P2 does not want to receive any more signals from the 2PC Signal Set.

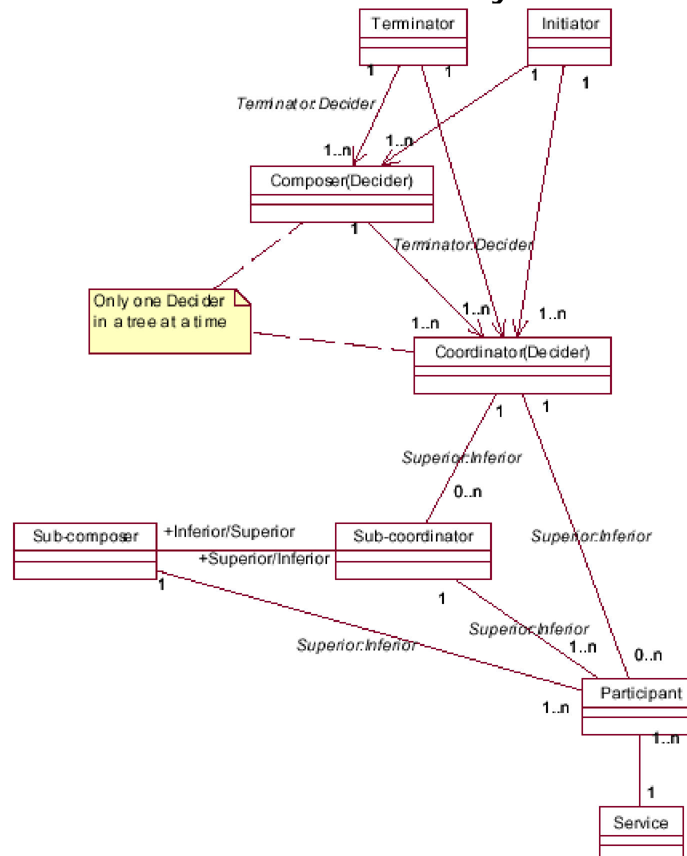
§ Therefore the ***get_outcome()*** method can be called to return the final outcome of the Signal Set to the AC, this would then in turn be returned to a higher level application. The transaction in this scenario has failed.

Further execution scenarios are contained within the appendix of this document. For more information on the Activity Service refer to [OMG00], the Activity Service specification.

3.2 – A Closer look at The Business Transaction Protocol

To continue from what was illustrated in Chapter Two, a breakdown of the constituent components of BTP is needed to illustrate the implementation of this project. Shown in the Figure below is an overview of the roles involved in BTP. These roles determine how the overall protocol fits together.

Figure 10: An overview of the key roles in BTP



BTP is designed to run over the architecture of the Internet, which is a loosely coupled, unreliable, untrustworthy environment; it aims to coordinate the work in the transaction into a consistent business state change. The following sub sections describe the different roles involved in BTP.

3.2.1 – Superior

A Superior is an abstraction for a coordinator, it accepts enrolments from an Inferior, establishing a Superior:Inferior relationship. One Superior can have relationships with many inferiors. The Superior determines the outcome applicable to the Inferiors, which are enrolled within it.

When the commit protocol is initiated the Superior issues prepare to all enrolled Inferiors, the Inferiors then reply as to whether they can prepare their work. A decision is then made and is propagated to the Superiors' enrolled Inferiors. A Superior can be either Atomic or Cohesive. The two different types of Superior apply different semantics to its enrolled Inferiors.

3.2.1.1 – BTP Atoms

A BTP Atom is an atomic unit of work, such that either all of the work is completed; or none of the work is completed. An Atom can be regarded as similar to a traditional Atomic transaction in a tightly coupled system, in the sense that a global decision is applied to all participants enrolled within an Atom. This is achieved by applying the two-phase commit protocol to the enrolled Inferiors to receive a constituent decision. Each atom can manage one or more Inferiors, and the Inferiors act on behalf of the Services to either confirm or cancel the work offered by the Service. An example of an Atom at work is illustrated in section 3.3.1.

3.2.1.2 – BTP Cohesions

A BTP Cohesion is a unit of work such that different outcomes can be applied to different Inferiors (participants) enrolled in the Cohesion. The outcomes that are applied to the Inferiors enrolled in the Cohesion are made with an application intervention and it is up to the user of the Cohesion to decide which subset of Inferiors to confirm and which to cancel. This allows a finer level of control over the coordination of Inferiors in the transaction, allowing an element of choice and a delayed decision by the end user to obtain the desired result. A Cohesive coordinator has a more complex role than an Atomic Coordinator. An example of a Cohesion at work is demonstrated in section 3.3.2.

3.2.2 – Inferior

An Inferior is responsible for applying the Outcome (received from a Superior, either confirm or cancel) to a set of operations. An Inferior can only be enrolled with a single Superior.

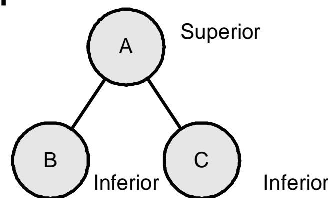
When an Inferior receives the Prepare message, it replies to the Superior it is enrolled with as to whether or not it can prepare the work. If the Inferior can ensure that a Confirm decision can be applied to its associated operations and it can persist the information then it will reply with Prepared. If it cannot guarantee this then a Cancelled message is sent to the Superior. An outcome is then sent to the Inferior notifying it of

the decision of the Superior; this decision will have different semantics depending on whether the Inferior is enrolled with an Atomic or a Cohesive Superior.

3.2.3 - The Superior:Inferior relationship

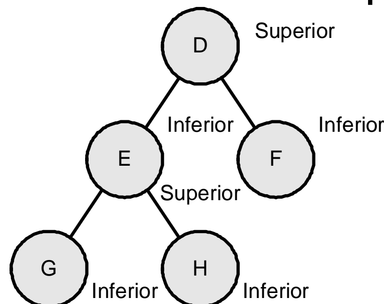
A business transaction tree can be arbitrarily complicated, and can involve a complex combination of Superior:Inferior relationships. The example illustrated in the figure below shows two independent Inferiors (B and C) which are enrolled with a Superior (A), The Inferiors B and C are completely independent of one another, they are only linked via C which applies an outcome to the Inferiors, this will be the same for an Atom Coordinator or can be different if the Superior is a Cohesive coordinator.

Figure 11: A Superior with two enrolled Inferiors



Superior:Inferior relationships can be nested within one another; E is an Inferior to D but a Superior to G and H. If node E is an atomic superior to G and H then it is known as a ***sub coordinator***; if it is cohesive then it is known as a ***sub composer***. In this case E will collect information on its enrolled Inferiors G and H before reporting back to its own Superior D.

Figure 12: Superior:Inferior relationships can be nested

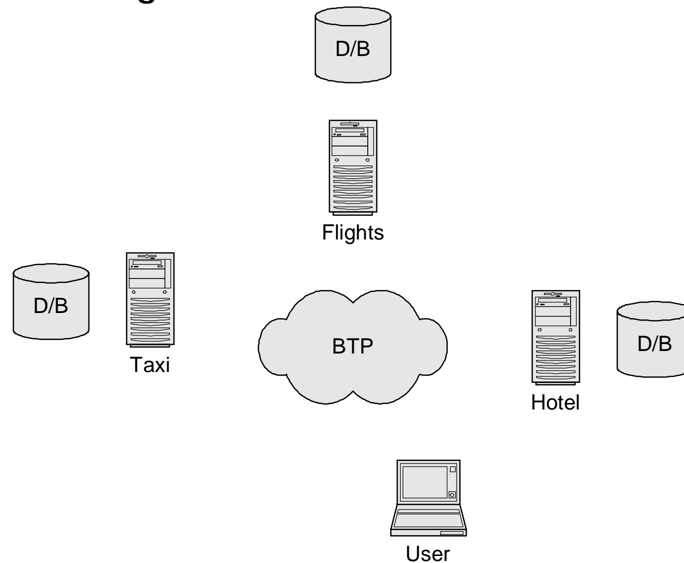


There are no fixed limitations on how deep the transaction tree can go, it can consist of an arbitrary number of Superior:Inferior relationships and can result in some very complex interactions. An Initiator will request a factory which will return a decider, this will either be a new top level decider which will be a new top level BTP transaction, or a sub-coordinator or sub-composer if this operation is performed within the scope of a current BTP transaction context.

3.2.4 - Services

A service contains the business logic to perform some kind of task for a user. For example if a user wanted to book a taxi firstly a reference to the taxi service must be obtained, once this had been done a remote method (e.g. bookTaxi()) can be called, with some kind of restrictions placed by the user. The service will then will query a back end database according to the users requirements and if those requirements can be met will enrol a participant with the coordinator of the transaction. When the commit protocol begins the coordinator of the transaction will talk to the enrolled participants.

Figure 13: BTP interactions



3.2.4 – Participant

A participant is an Inferior that is application specific. When the word participant is used in relation to BTP it is referring to an application specific Inferior that is updating some kind of resource on behalf of a service. So an Oracle database would have a different participant to a Taxi booking service, its completely application specific. A participant has exactly the same interface as the Inferior; all that it defines is what happens when it receives the relevant BTP messages, and its up to the service providers to specify this and implement the logic inside the participant. A participant is responsible for determining whether a prepared condition is possible when the two-phase commit protocol is initiated, and for then applying the global decision to the resource.

3.2.5 – Decider

A decider is a Superior that is not also the Inferior in a Superior:Inferior relationship. It can be seen as the top node in the BTP Transaction tree and receives its requests from

the Terminator as to the desired outcome of the business transaction. A decider receives commands to Prepare/Cancel some or all Inferiors and to confirm or Cancel the transaction, it will report back to the Terminator the result of the business transaction.

There are two types of Decider a ***Coordinator***, which is an Atomic Superior (has the properties of an Atom) and a ***Composer***, which is a Cohesive Superior (has the properties of a Cohesion).

3.2.6 – Terminator

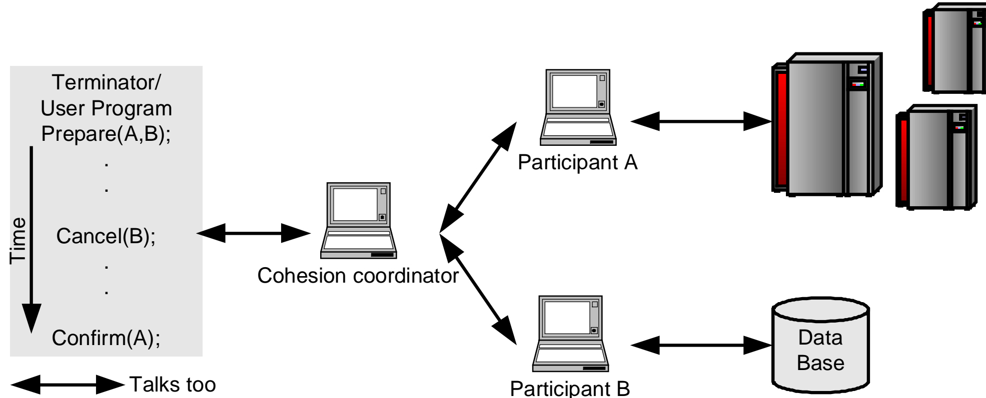
The terminator talks to the top node in the transaction tree, the Decider. A Terminator is usually an application element (e.g. A User who wishes to create a transaction, for example booking a holiday as shown earlier). The Terminator initiates all communication between the Terminator and the Decider. A Terminator will talk to a Coordinator (Atomic) in a different way to a Decider (Cohesive), for example the Terminator may ask the Composer to prepare some of its inferiors, this is not allowed in a Coordinator, as this is an Atomic unit. The Terminator can ask the Decider to cancel (all or part of) the business transaction.

A Terminator is an application element, for example the organization of a night out. The user will create a Cohesion, connect to some kind of services (taxi booking, hotel service etc.), from here the services would invoke the business logic and enrol a participant with the Cohesion. Its then up to the user what to do, whether to prepare/cancel some (or all) of the Inferiors or to Confirm or Cancel the Transaction.

3.2.7 – BTP exposes both phases to the user

In a traditional transaction a user will simply place some work that needs to be transactional inside a start and an end point, the participants are then invoked with the two phase commit protocol and the work is either Committed or Aborted as an Atomic unit. BTP allows both phases of the commit protocol to be seen and controlled by the end user. The protocol allows the user to independently prepare, confirm or cancel participants enrolled in a transaction. This finer level of control is needed because BTP transactions are designed to run for long periods of time, participants can be prepared separately with large intervals of time between them before a final decision (commit or cancel) is made. Shown in the figure below is an example of how a Terminator would control the flow of the transaction.

Figure 14: Both phases of the Commit Protocol



Seeing both phases allows a much finer level of control over the transaction than that offered by traditional Atomic transactions.

3.2.8 – Opinion from every direction

BTP allows asynchronous communication between a Superior:Inferior relationship, communication is possible in both directions, both from a Superior to an Inferior and from an Inferior to a Superior. In traditional transaction systems the coordinator of the transaction has overall say on the outcome of the transaction, however in BTP the user has no real control over the resources that are taking part in the transaction, nothing has been guaranteed and they are operating in the loosely coupled environment of the Internet. Therefore BTP listens to the services that it is interacting and tries to accommodate there needs. Normally the Superior, which passes down messages to any enrolled Inferiors, initiates communication, however BTP allows Inferiors to make autonomous decisions to apply the associated operations without waiting for the messages from the Superior. An Inferior can make an autonomous decision to prepare, confirm or cancel work early without waiting for the relevant message from its BTP Superior.

Extra information can be embedded within messages that are sent to the coordinator, this extra information is represented as Qualifiers. A Qualifier is a way of inserting extra information into a message that is sent between a Superior:Inferior relationship, there are three main qualifiers that are used in BTP:

Transaction Time Limit: This qualifier is used by a user of the BTP API, it suggests a maximum length of time that the transaction is likely take. A participant can use this to determine whether the coordinator has crashed if no messages are received.

Inferior Timeout: A participant can return this qualifier to the coordinator, with the prepared message. It indicates how long the participant will remain prepared for and what action it will take if that time is exceeded.

Minimum Inferior Timeout: This qualifier is passed from the coordinator to a participant, it states the minimum time the participant must take part in the transaction. If the participant cannot comply with this then it will return Cancelled.

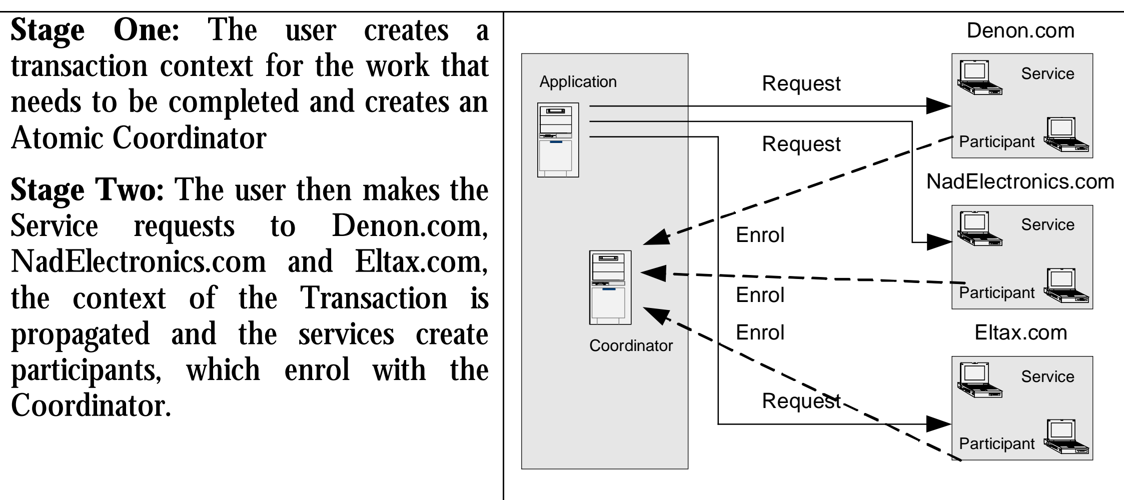
In this way BTP listens from all directions and tries to accommodate the needs of not hjust the user but also the participants in the transaction. This topic is addressed in more detail in section 4.2.3.

3.3 – Real World Example using BTP

To take a real world example of BTP and to apply the concepts and terms defined in this chapter, lets look at a real example, the purchasing of a stereo over the internet, various vendors and retailers have offered services to users who wishes to purchase goods from them.

3.3.1 - Multiple Party Atomic Transaction

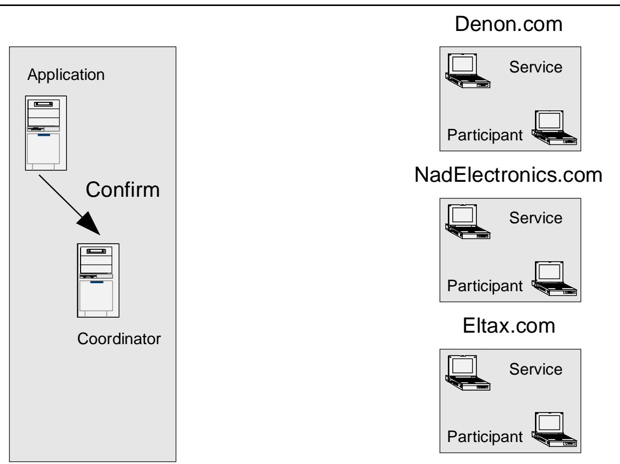
Lets consider the situation where a user wants to purchase a separates stereo online. The user already knows what components they want and this consists of an Amp (from Denon), CD Player (from NADElectronics) and a pair of speakers (from Eltax). The user wants the entire stereo to be purchased as an Atomic unit, either all of it is purchased or none of it is.



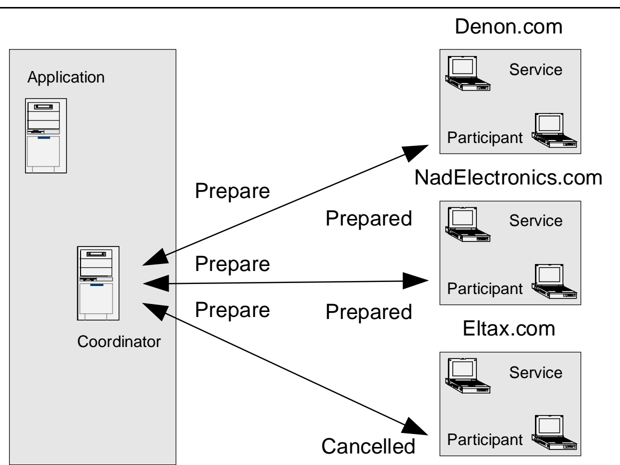
Stage One: The user creates a transaction context for the work that needs to be completed and creates an Atomic Coordinator

Stage Two: The user then makes the Service requests to Denon.com, NadElectronics.com and Eltax.com, the context of the Transaction is propagated and the services create participants, which enrol with the Coordinator.

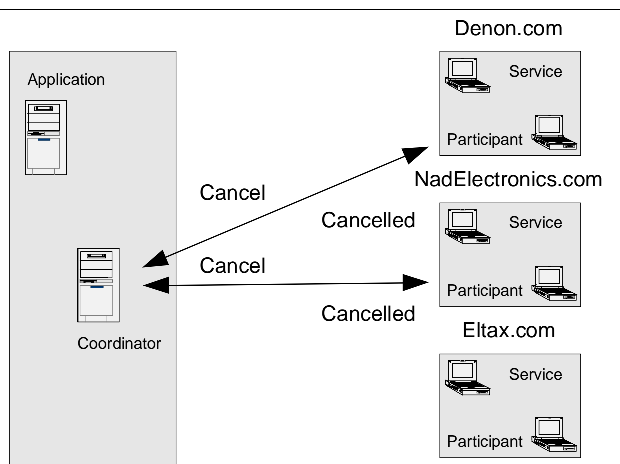
Stage Three: Now the three services have enrolled the participants in the business transaction, the user asks to Confirm the transaction, which initiates the two phase commit protocol.



Stage Four: Two phase commit is initiated; prepare is sent to the participants in the transaction, Denon, NadElectronics and Eltax. Once the participant receives the prepare message they must reply as to whether they can Prepare their work.

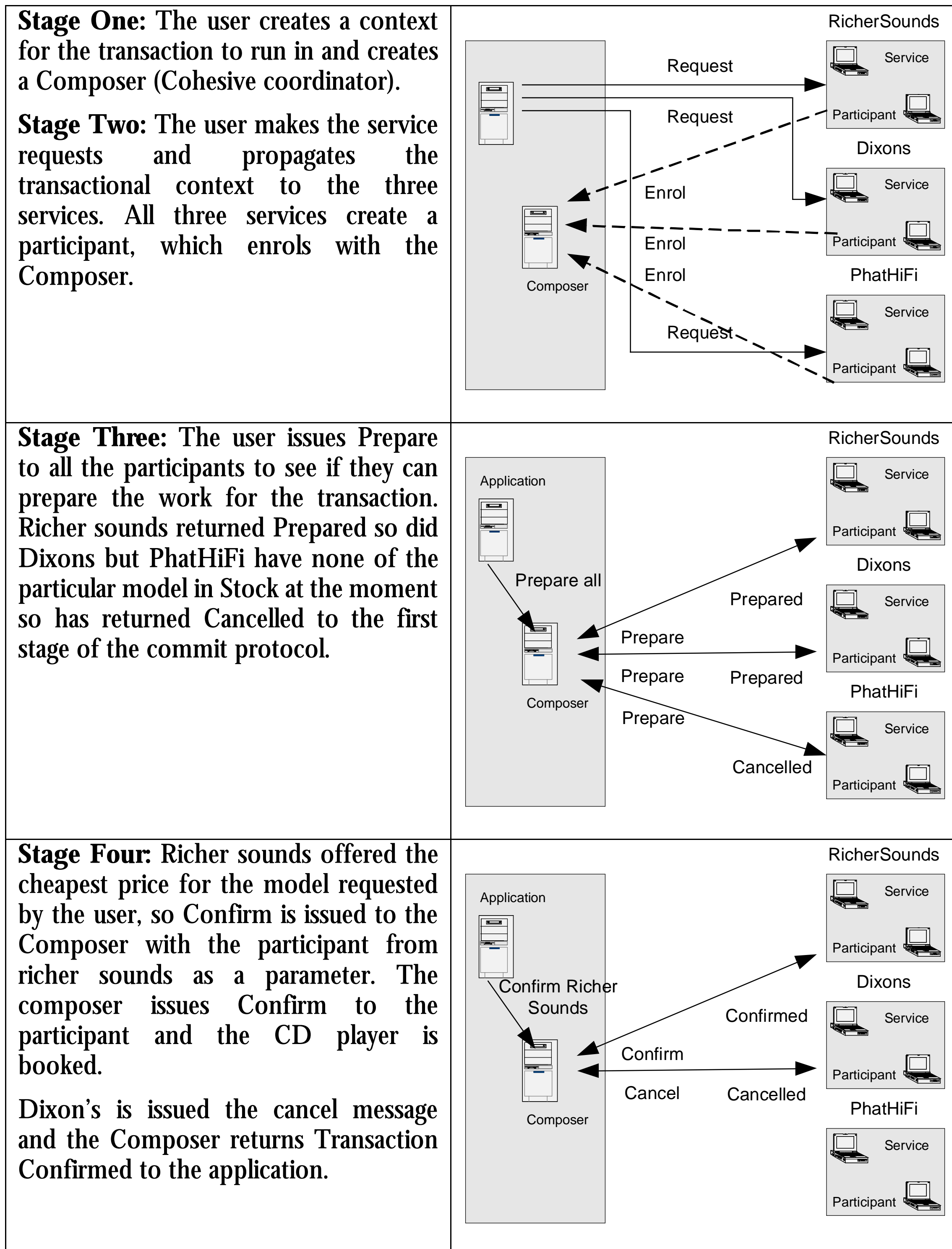


Stage Five: Denon and NADElectronics replied with Prepared, how ever Eltax (the speaker company) replied with Cancelled and because the transaction is executing in the scope of an Atom then the other two participants must be cancelled. The Coordinator informs the Application that the Transaction has been cancelled.



3.3.2 - Cohesion Example

Consider now an example where the user wants to buy a certain model of CD player from NADElectronics and wants to shop around for the cheapest price on line.



Stage	Diagram
Stage One: The user creates a context for the transaction to run in and creates a Composer (Cohesive coordinator). **Stage Two:** The user makes the service requests and propagates the transactional context to the three services. All three services create a participant, which enrols with the Composer.	
Stage Three: The user issues Prepare to all the participants to see if they can prepare the work for the transaction. Richer sounds returned Prepared so did Dixons but PhatHiFi have none of the particular model in Stock at the moment so has returned Cancelled to the first stage of the commit protocol.	
Stage Four: Richer sounds offered the cheapest price for the model requested by the user, so Confirm is issued to the Composer with the participant from richer sounds as a parameter. The composer issues Confirm to the participant and the CD player is booked. Dixon's is issued the cancel message and the Composer returns Transaction Confirmed to the application.	

CHAPTER FOUR

BTP ON THE ACTIVITY SERVICE

To make up a coordination model using the Activity Service you need to specify the Signal Sets, the signals which reside within them, the Actions and how these components are made use of within the scope of an Activity. This section will look at how BTP maps onto these different components and how an implementation of the protocol was designed and implemented.

4.1 – Interfaces to BTP

The user of this implementation of BTP needs to be abstracted away from the underlying details of the Activity Service; the user need not know anything about Signal Sets, Signals and Actions: all they are provided with is an interface to the different roles of BTP. However for the projects sake it's necessary to take a look under the covers to see how the different components of the Activity Service interact to form an implementation of BTP. Two main interfaces are provided for BTP; these are relatively simple and are broken down into the roles of Inferior and Superior.

4.1.1 - Inferior Interface

The Inferior interface would be used to construct an application specific participant. Inside the method bodies contain what happens to a resource when they are called, this would change from application to application and it is up to the provider of a service to specify this. An Oracle database would implement the Inferior interface differently to a taxi booking service. The Inferior interface contains the following methods:

Prepare: This method requests a reply to the question, can you prepare your work for the transaction? The Inferior will reply with either true (which represents Prepared) if it can ensure that either a confirm or cancel decision can be applied, or if it cannot maintain this ability it will reply with false (which represents Cancelled).

If the Inferior has itself got enrolled Inferiors then the message must be passed to all these enrolled Inferiors, the Inferior which is enrolled in the Superior:Inferior relationship will then reply with a conscious opinion. Either Prepared if all enrolled Inferiors reply Prepared or if one of them replies Cancelled it will reply with Cancelled.

Confirm: This method is called on an Inferior that has returned Prepared to the first stage of the commit protocol. Once this method has been called the Inferior can confirm the effects of the transaction and update the resources etc. on behalf of the service.

Cancel: This method can be called on an Inferior at any time before Confirm has been called, the inferiors' work will be cancelled and the Inferior will not be referenced again.

Shown below is an example of a class which extends the Inferior interface, a user of this API must fill in what happens to a resource when each of the methods are called. In this example the class that implements the Inferior interface is a Participant for a Cinema.

```
public class CinemaParticipant extends Asynchronous implements Inferior
{
    public boolean prepare() throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed
    {
       /*
        Business Logic
        What to do when the Prepare method is called, return true to indicate Prepared
        or false to indicate Cancelled.
        */
        return false;
    }
    public void confirm() throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed
    {
        /*
        Business Logic
        What to do when the Confirm method is called.
        */
    }
    public void cancel() throws GeneralException, InvalidInferior, WrongState, Hazard,
Mixed
    {
        /*
        Business Logic
       What to do when the Cancel method is called
        */
    }
}
```

4.1.2 - Superior Interface

The Superior interface determines the outcome that is applicable to its enrolled Inferiors; a Superior can be either Atomic or Cohesive. This interface is never handled directly by the user, the Atom and Cohesion interfaces (discussed in the next sub section) implement the behaviour defined here. The interface to a Superior is also a relatively simple interface; it provides methods too:

Enrol: An Inferior enrols with a Superior. When enrol is called on a Superior, a reference to the Inferior is supplied and a Superior:Inferior relationship is established.

The Superior can now issue messages to the enrolled Inferior when the commit protocol is initiated.

Resign: An Inferior can end the relationship with the Superior at certain points in the proceedings by calling the Resign method, this will depend on whether the Superior is Atomic or Cohesive. Other Inferiors can still enrol with the Superior.

Request Inferior Statuses: This method returns the statuses of all Inferiors that are currently enrolled with the Superior.

A Superior can provide either be Atomic or Cohesive to its enrolled Inferiors. This logically breaks down into two more interfaces, an interface for an Atom and one for Cohesion. These interfaces will be the main point of contact with a user of the BTP API's.

4.1.3 - Atom Interface

An Atom implements the Superior and Inferior interfaces; it provides Atomic behaviour for all of the Inferiors enrolled within it and provides the functionality of the Inferior and Superior interfaces. Inferiors enrol with an Atom, one Atom can have many enrolled Inferiors, however an Inferior can only be enrolled with one Atom, this is a 1:Many relationship. Once they enrol they establish a Superior:Inferior relationship, the Atom being the Superior.

Once the relationship is established messages can be issued to the enrolled inferiors when the commit protocol is initiated. Once the commit protocol is initiated (by the invoking the Prepare method) no more Inferiors are permitted to enrol within the Atom. When an Atom is created it starts a new top level Activity, this means that its independent of any other running Activities and is not running within any other scope. The Coordinator that is created, stored and is referenced when methods are called on the Atom.

4.1.4 - Cohesion Interface

Cohesions provide a finer level of control over enrolled Inferiors; the strict ACID properties are relaxed. Atoms enrol with Cohesions. The interface is slightly more complicated than that of an Atom. It provides operations to:

4.1.4.1 – Prepare Inferiors

This method prepares the work of some or all of the Atoms that are enrolled within a Cohesion, by calling the Prepare method on each of them in turn, as long as they have not already returned Prepared, Resign or Cancelled. Atoms are still permitted to enrol within the Cohesion after this method has been invoked.

This method can be called in two different ways:

Null parameter: If this method is called with a null parameter then the Cohesion will attempt to prepare all enrolled Atoms, by calling the Prepare method on each of them in turn.

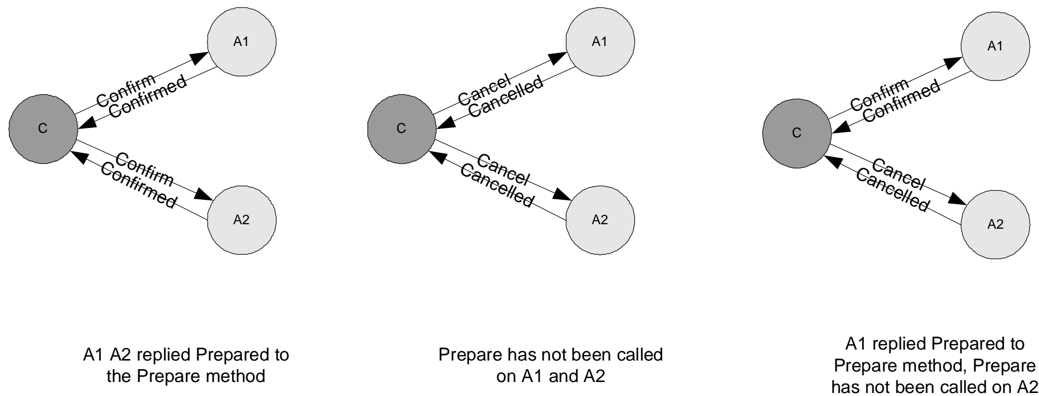
List of Inferiors: If the method is called with a list of Inferiors then the Cohesion will prepare only the Atoms identified in the Inferiors list. In both cases the method will return a list of Inferior Statuses, so that the user of the API can see the result of the first stage of the commit protocol and see what action to take next.

4.1.4.2 – Confirm Transaction

This is the method that is used to request the confirmation of the business transaction; it can also be called in two different ways, which apply different behaviour:

Null parameter: If the parameter is null then this implies that all enrolled Atoms make up the Confirm Set. Each individual enrolled Atom will be treated differently. When called with a null parameter this method tries to confirm what it can. If the Prepare method has not yet been called on the Atom then Cancel will be called instead. If the Prepare method has already been called on the Atom and it replied with Prepared (true) then the Confirm method will be called on the Atom. If the Atom replied with Cancelled (false) to the Prepare method then nothing needs to be called on this Atom, as it is already in a valid end state. Shown in the Figure below is a demonstration of the confirm transaction method in different scenarios, the Cohesion has two enrolled Atoms.

Figure 15: Confirm Transaction scenarios



List of Inferiors: If the Confirm Set is present then the Atoms identified in the parameter list make up the Confirm set (The Atoms that will be confirmed). For a Confirm decision to be made all the Atoms identified in the Confirm Set must have replied Prepared to the first stage of the commit protocol (the Prepare method). If this is the case then the Cohesion will reply with Transaction Confirmed to the Terminator. Otherwise a Cancel decision will be made and the Cohesion will reply with Transaction Cancelled to the Terminator.

If the prepare method has not been called on any Atoms in the confirm set then this must be done first. The outcome of this is noted and if any of the Atoms replied with Cancelled then a global Cancel decision will be made. Once Prepare has been called on all the Atoms identified in the Confirm Set, a decision must be made to confirm or cancel the Cohesion. If all the Atoms identified in the Confirm set have replied Prepared to the first stage, Confirm will be issued to them. If one or more Atoms in the confirm set have replied Cancelled to the first stage then all Atoms in the confirm must be cancelled; this is done by calling the Cancel method on the Atoms that returned Prepared to the first stage of the commit protocol. Atoms that are not identified in the Confirm set shall be cancelled.

Two examples are shown below, the Confirm set contains two Atoms that need to be confirmed. The prepare method has not been called on A1, so this must be done first to see whether an overall confirm or cancel decision is to be made. In scenario one the Atom replies with Prepared and hence the business transaction is confirmed. In scenario two the Atom replies with Cancelled and hence the remaining Atom in the confirm set must also be cancelled.

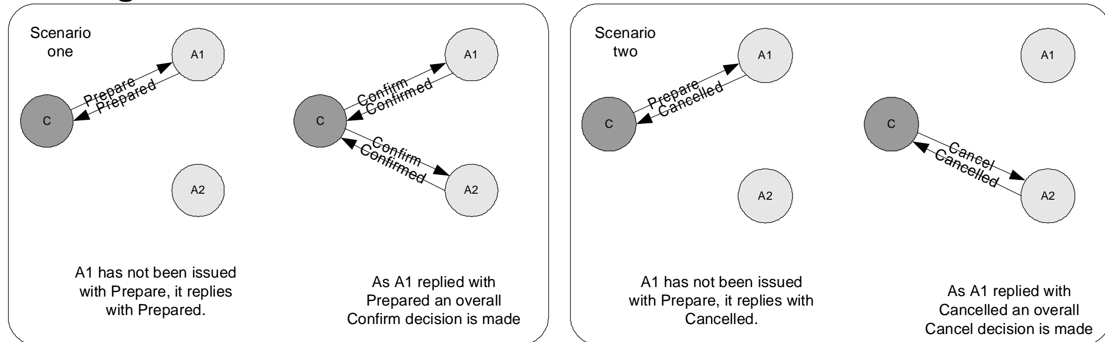


Figure 16: Confirm Transaction with an Inferiors list

4.1.4.3 – Cancel Transaction

This method can be called at any time other than when Confirm Transaction has been called on the cohesion. The business transaction is cancelled and this is propagated to all remaining Atoms (which will propagate to any enrolled Inferior(s)) by issuing the cancel method to them. Once this method is called, no more Atoms will be permitted to enrol with the Cohesion.

4.1.4.4 – Cancel Inferiors

This method calls the cancel method on all or some enrolled Atoms. Once this method is called new Atoms are still permitted to enrol with the Cohesion. This method can also be called in two different ways

Null parameter: If the parameter is null then all Atoms that are enrolled with the Cohesion will be cancelled. However, new inferiors may be permitted to enrol with the Cohesion, this is a very important difference a Cohesion and an Atom, as when an Atom begins its commit protocol no more Inferiors are permitted to enrol.

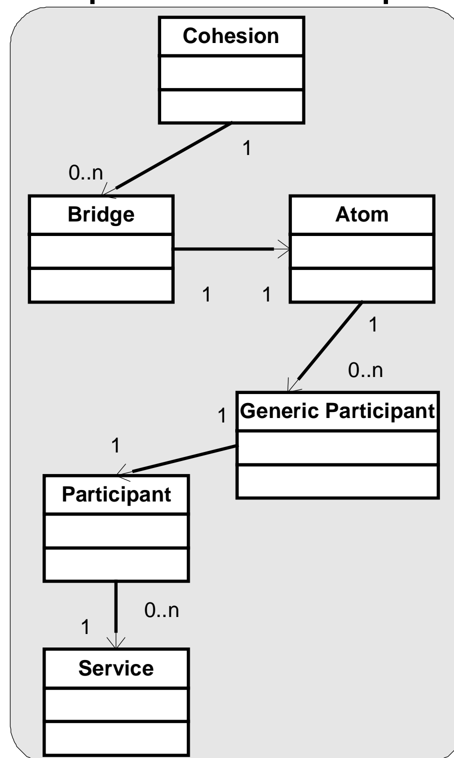
List of Inferiors: If the inferior list is not empty then the Atoms identified in this list will make up the list of Atoms to be cancelled. The Cancel method will be called on these Atoms, all others will remain unchanged. New Atoms will still be permitted to enrol with the Cohesion.

4.2 – Architecture of the Solution

4.2.1 – Overall View

Shown below is a diagram, which describes the overall structure of the implementation. Each component will then be broken down and looked at in turn and the interactions explained. Cohesion and Atom on the figure represent the interfaces that have just been defined.

Figure 17: Components of the implementation



4.2.2 - How do Superiors and Inferiors talk to each other?

As shown in the previous chapter, Actions register with Signal Sets, receive signals and return an outcome, which ends up back at the Signal Set. When the Activity Coordinator asks for the next signal the logic of the Signal Set decides which signal (if any) to send based on the latest outcome it has received.

To use the Signal Sets that have been defined for this project requires registering Actions with them. A user of the BTP API's doesn't need to know about Actions in regard to the Activity Service, these details should be abstracted away; so two types of generic Actions have been created. These actions never change and the user doesn't need to know they exist. They are effectively protocol translators, taking Activity Service invocations and converting them to BTP and visa versa, making the use of the Activity

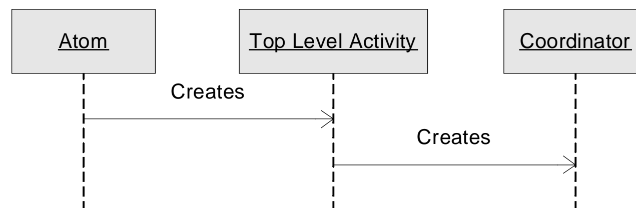
Service completely transparent to the user, all the user is presented with are the interfaces defined in Section 4.1.

It's important to look at these details, as they are the key to how the underlying complexities of the Activity Service have been abstracted away from the user of this API.

4.2.2.1 – Enrolling an Inferior with an Atom

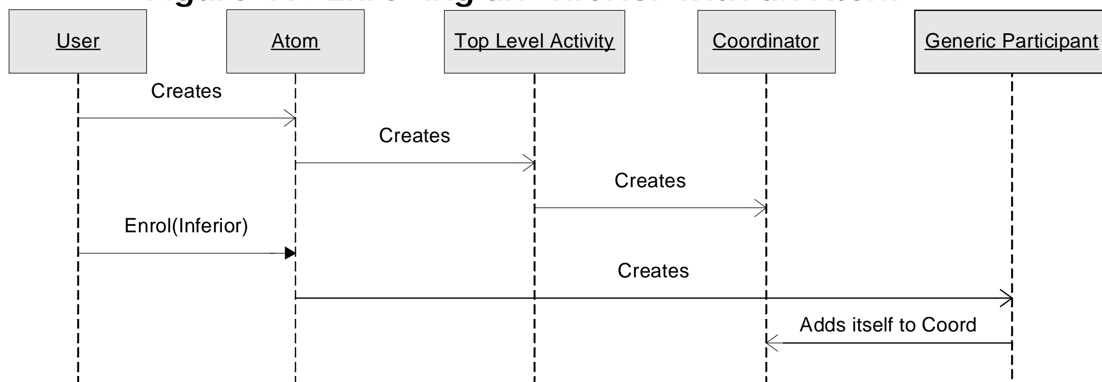
The Generic Participant is a class that implements the Action interface. It sits between the Atom and the application specific participant (Inferior) that the user has created. When the Atom constructor is called, a new top level Activity is created, and the reference to the Coordinator is made local, so when ever calls are made on the Atom the local coordinator is referenced. This is illustrated in the Figure below.

Figure 18: Creation of an Atom



The user of the API only explores BTP roles. When an Inferior enrols with an Atom the following takes place:

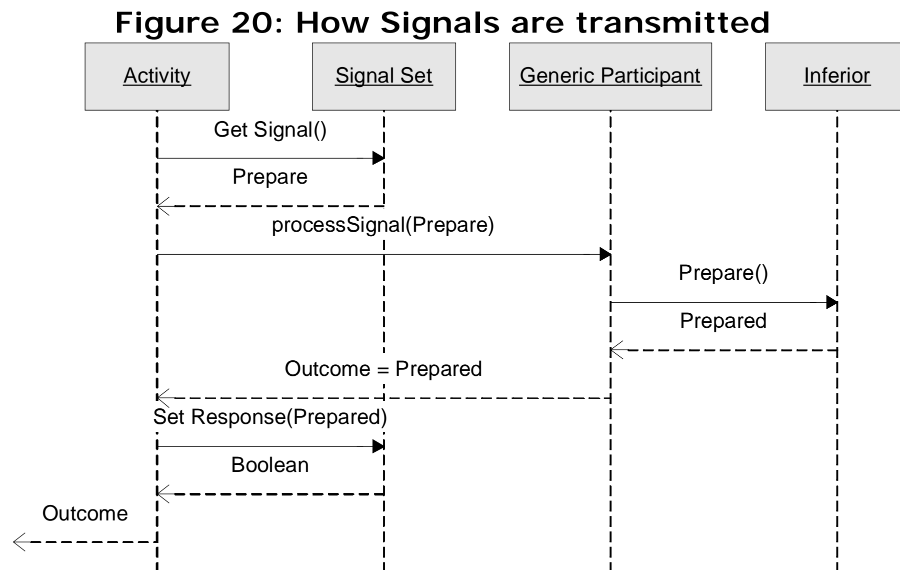
Figure 19: Enrolling an Inferior with an Atom



To elaborate on the above diagram, when a new Atom is constructed it creates a Top Level Activity and a reference to a local Activity Coordinator is stored. A user can then enrol an Inferior with the Atom (by calling the enrol method, passing the newly created Inferior as a parameter). When this method is called a new Generic Participant (Action) is created (There is a one to one mapping a Generic Participant and an Inferior), which

registers itself with the local coordinator stored within the Atom. A reference to the Inferior is passed to the Generic Participant(Action), in this way it appears to the Inferior as though it is interacting with an Atom Coordinator when in fact the messages are being passed through the Generic Participant that has just been created. The figure shown below illustrates what happens when an Atom talks to an enrolled Inferior:

Figure 20: How Signals are transmitted



To elaborate on the above, the Activity Coordinator created when the Atom is initialised passes the Signals (returned by the get_signal() method) to the Generic Participant by calling the process_signal() passing the signal as a parameter. The Generic Participant then makes a call on the reference it has to the participant, mapping the Activity Service invocations to their BTP equivalents. The result of this call (if any) is passed back to the Generic Participant, which returns this to the Signal Set (via the Coordinator) as an Outcome (with the response embedded in it) through the set_response() method that is called on the Signal Set. The logic of the Signal Set then takes over and when the Activity Coordinator asks for the next signal the outcome of the last signal is used to see which (if any) Signal should be passed to the registered Actions.

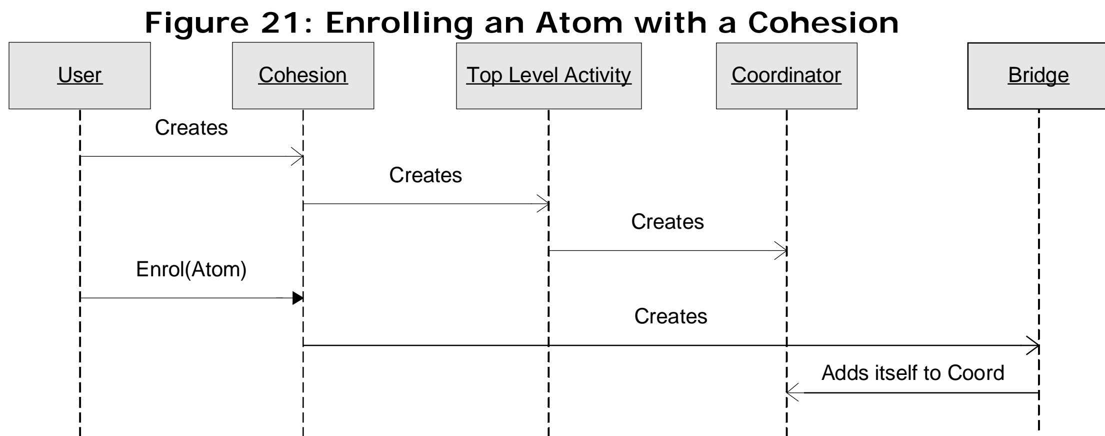
In this way the user defined participants have no knowledge that the Inferior is in fact talking to a class in the middle, which is an Action registered with a Coordinator, enabling the underlying details of the Activity Service to be hidden from the end user.

4.2.2.2 – Enrolling an Atom with a Cohesion

The same concept is applied to the Bridge class; which also implements the Action interface. The bridge sits between the Atom and the Cohesion. When a Cohesion is created, a top level Activity begins and a reference to the Coordinator is stored. Method

invocations on the Cohesion are referencing the local Activity Coordinator of the Cohesion. Atoms enrol with Cohesions, when enrol is called on the Cohesion an Atom is passed as the argument to the function, and in turn a new Bridge class (Action) is created which registers itself with the Activity Coordinator of the Cohesion. The Atom can now be referenced via the bridge class. This is demonstrated in the figure below:

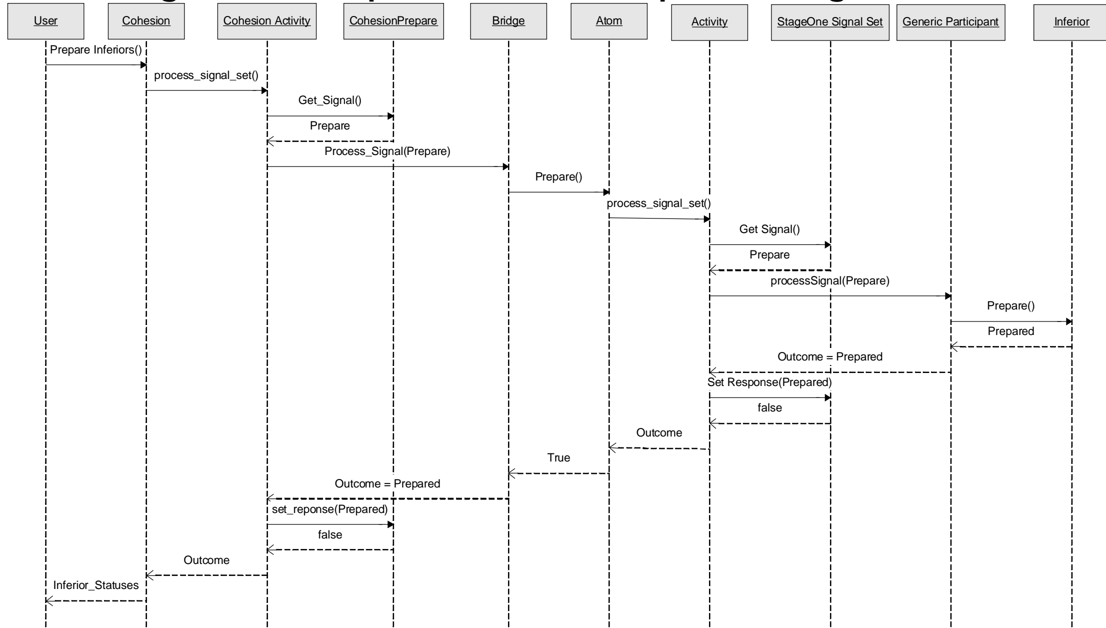
Figure 21: Enrolling an Atom with a Cohesion



When function calls are made on the Cohesion (e.g. prepare all enrolled Inferiors) the Signals (that are returned by the get_signal() method) are passed to the bridge, which contains the underlying logic to take the appropriate action in response to the Signal. It will then make the appropriate function call on the Atom class (following the process described in Section 4.2.2.1) and returns the outcome to the bridge class, which returns this to the Signal Set which the bridge is registered too. When the Activity Coordinator asks for the next signal the outcome that has just been returned can be used to make the decision of the which signal (if any) is to be sent next.

Demonstrated in the figure below is the sequence of function calls that take place when a user calls Prepare Inferiors on a Cohesion which has one enrolled Atom which in turn has one enrolled Inferior. More detailed definitions of the function calls can be found in the Activity Service specification [OMG00].

Figure 22: Prepare Inferiors sequence diagram



The difference between the Bridge and the Generic Participant is the underlying logic contained within the classes. Shown below are the methods that must be implemented to construct an Action that can be registered with an Activity Coordinator. Although they both implement the Action interface the constructors of the class are different, and the logic contained within the process_signal() methods is different. The Generic Participant contains logic in regard to the enrolled Inferiors (e.g. maintaining the Atomic property) and the Bridge class contains logic to coordinate the enrolled Atoms and is a more subtle way of manipulating enrolled Inferiors.

```
public class x extends  org.omg.CosActivity . ActionPOA
{
    public x()
    {
        //Constructor
    }
    public final synchronized Action getReference ()
    {
       //Returns a Reference to the Action
    }
    public Outcome process_signal (Signal sig) throws ActionError, SystemException
    {
       //Returns an Outcome when a Signal is received
    }
    public synchronized void destroy () throws AlreadyDestroyed, SystemException
    {
        //Destroys the Action
    }
}
```

4.2.3 - Asynchronous Behaviour

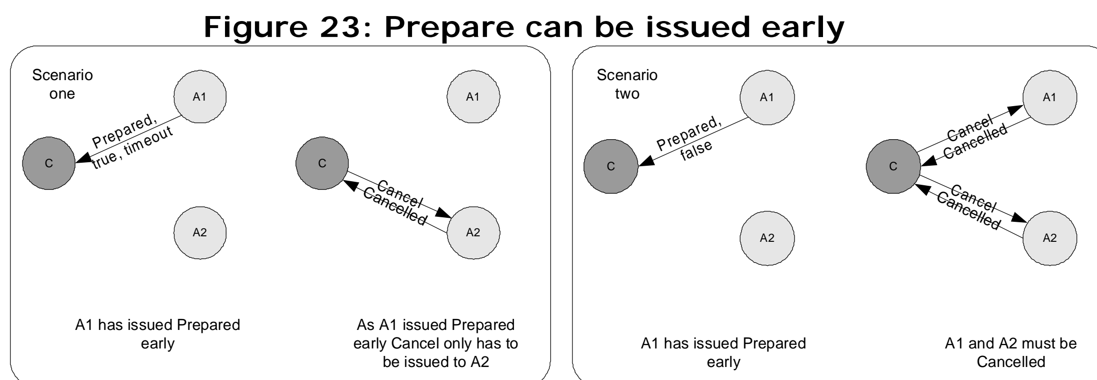
As demonstrated in Section 3.2.8, BTP allows Asynchronous communication between a Superior:Inferior relationship. Normally The Superior initiates the communication and

passes the relevant messages to the Inferior, however enrolled Inferiors can make autonomous decisions to prepare, confirm or cancel their work early without receiving the relevant message from their enrolled Superior, this is a perfectly legal thing for a BTP Inferior to do.

4.2.3.1 – Inferiors can Prepare early

An Inferior can issue Prepared to the coordinator early, without waiting for the Superior to call the Prepare method. An Inferior will do this if it does not need to wait until it receives the message to prepare the work for the business transaction. Shown below is an example of how Prepared could be issued early by an Inferior. A Qualifier is used in the prepared message.

Figure 23: Prepare can be issued early



In both scenarios one and two the Superior has two enrolled Inferiors and decides to make an overall Cancel decision. In Scenario one Inferior A1 issued Prepared to its Superior early, included in this message is a parameter known as, 'default is cancel', this is a Boolean value that if set to true means that if the Inferior doesn't receive a Confirm message then it will Cancel its operations. In the case shown in scenario one, a timeout is also included in a qualifier, so if a Confirm message is not received within the allotted time (indicated by the timeout value) then the Inferior A1 will autonomously cancel the associated operations. However if the Superior decides upon an overall Cancel decision then it does not need to bother A1 again, only A2 needs to be cancelled. So in the example the qualifier was set to true and the overall decision made by the Superior was Cancel, therefore only Inferior A2 needs to be cancelled.

In scenario two, this Boolean value is set to false, this means that no matter what the overall decision is the Inferiors wants to see the decision whether it's an overall Confirm or a Cancel.

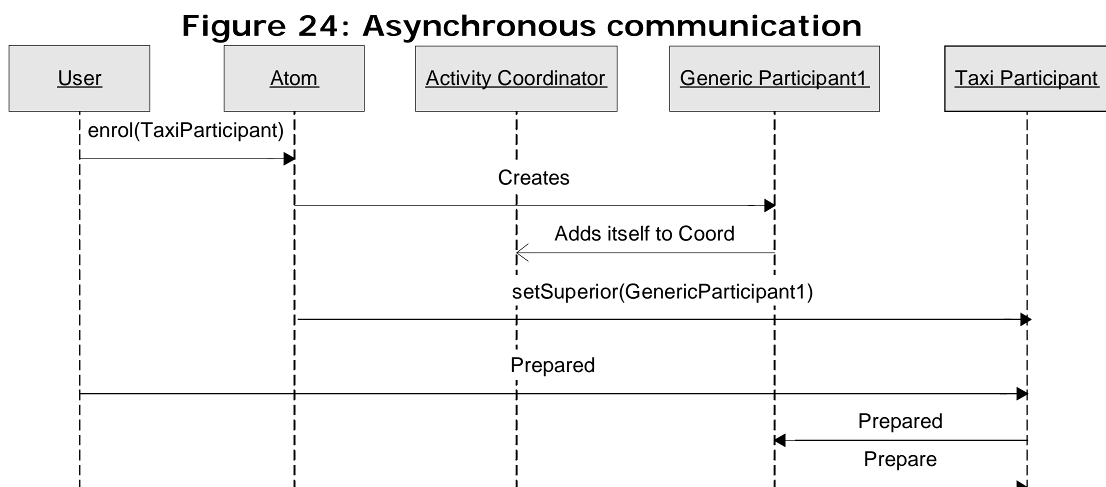
4.2.3.2 – Inferiors can Confirm or Cancel early

Cancelled and Confirmed can also be issued early from an Inferior to a Superior. If Confirmed is issued before the Confirm message is received, or after a Cancel message has been sent then this is regarded as an autonomous decision (the later will cause a Contradiction message to be sent) A Contradiction message is issued from the Superior to the Inferior when an autonomous decision is made which is contradictory to that of the Atom. (e.g. the Inferior makes an autonomous decision and replies with Confirmed however the decision applied to the Atom is Cancelled). The same is true of an Inferior making an autonomous decision and issuing the Cancelled message early. A parameter can be inserted into the Confirmed message, called 'Confirm received', this is a Boolean which is true if the Confirmed message was returned after receiving the Confirm message, it is otherwise false which indicates a autonomous decision.

So how does the BTP API created for this project handle the Asynchronous behaviour demonstrated in this section? As we have already seen in Section 4.1.1. a user who wishes to create a participant (Inferior) that is to take part in a transaction must implement the Inferior interface and fill in the method bodies as to what the Inferior must do when it receives certain messages, however to enable the Asynchronous communication the class must also extend the Asynchronous class.

If we look at the example shown in Section 4.2.2.1 again, there is actually more going on under the covers that needs to be looked at. Shown below is a UML Diagram showing the further steps that enable the Asynchronous communication between a Superior:Inferior relationships.

Figure 24: Asynchronous communication



To elaborate on the above figure, when the user calls enrol and passes it an Inferior (in this case TaxiParticipant) a new Generic Participant is created, which adds itself to the local Coordinator of the Atom as an Action. Once this has been done, a method, Set_Superior, can be called on the Inferior (this methods comes from extending the Asynchronous class), this method enables the Inferior to remember the reference to the Generic Participant, which is passed as a parameter to the function. This means that the Inferior can make invocations on the Generic Participant at the users discretion.

A service provider has the following methods available, Prepared, Confirmed and Cancelled. Similar methods are available on the Generic Participant. When these methods are called on the Inferior, the corresponding method is called on the Generic Participant. In the example shown in the above figure, when the user (normally the service provider) calls Prepared on the Inferior (meaning that the participant doesn't need to wait for the Prepare message) this in turn invokes the Prepared method on the Generic Participant. This in turn calls the Prepare method on the participant (Inferior), this does the application specific work contained in the Prepare function body of the participant. This means that when the commit protocol is finally initiated and this Action receives the Prepare message, it simply ignores it, as it has already prepared the work for the transaction earlier.

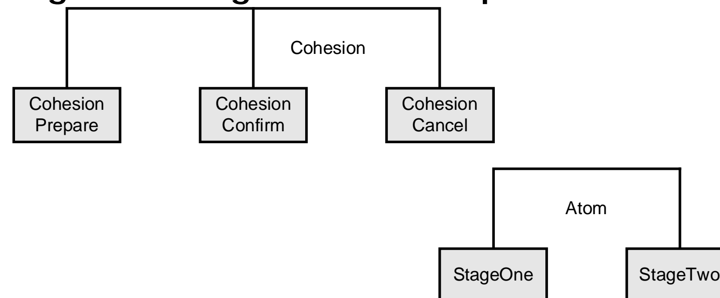
These methods allow Qualifiers to be inserted into them to implement the behaviour discussed earlier. However work to enable the full functionality discussed in the specification needs to be carried out and is discussed in the Future Work section.

In this way the user will see nothing of the underlying Activity Service details, all that needs to be done to emulate the behaviour of asynchronous messaging between a Superior:Inferior relationship is to fill in the function bodies of the Inferior class which implements the Inferior interface and extend the Asynchronous class.

4.2.4 – Signal Sets

There are five Signal Sets that have been used to implement BTP, two for the Atom and three for the Cohesion, as illustrated in the diagram below. A description of these Signal Sets follows:

Figure 25: Signal Sets to implement BTP



4.2.4.1 - Atom Signal Sets

StageOne: This Signal Set deals with the first stage of the Commit protocol: it issues Prepare to all the enrolled Inferiors (usually participants created by the Service) waits for the results and then applies the global rule to these results producing an Outcome. The Signals, possible outcomes and their meaning are shown in the table below.

Figure 26: StageOne Signal Set

Signals	Outcomes	Meaning
Prepare	Success	Every enrolled inferior has returned prepared to the first stage.
	Failure	Every inferior issued cancelled to the first stage.
	Failure with Rollback	At least one Inferior has returned Cancelled to the first stage, however at least one inferior has issued Prepared (meaning they need to be rolled back).

StageTwo: This Signal Set takes the outcome from the Stage One Signal Set and applies it to the enrolled inferiors. It will behave differently depending on what the outcome was from Stage One.

It will either: Issue Commit to all enrolled participants if the outcome was success from Stage One, do nothing if the outcome was failure (all inferiors returned cancelled to Stage One), or if the outcome is Failure with Rollback then Cancelled will be issued to

the Inferiors that returned Prepared to the first phase of the commit protocol. The Signals and their meanings are shown in the table below:

Figure 27: StageTwo Signal Set

Signal	Meaning
Confirm	Confirm the Atom, Issue Confirm to Inferior(s)
Cancel	Cancel the Atom, Issue Cancel to Inferior(s)

If the user has to drive both phases of the commit protocol, by issuing Prepare, Confirm and Cancel independently to the Atom the user must call the functions in the correct order, otherwise the relevant BTP Exception will be thrown, for example when the user tries to Cancel an Atom that has already Confirmed its work.

4.2.4.2 - Cohesion Signal Sets

CohesionPrepare, CohesionCancel and CohesionConfirm

These are relatively simple Signal Sets that sit on top of the existing infrastructure for an Atom. They are merely a signalling mechanism to initiate control of any enrolled Atoms. CohesionPrepare has one signal Prepare, CohesionCancel has Cancel and finally CohesionConfirm has the confirm signal. Inferiors of the Cohesion will be added to different Signal Sets at the relevant points in the life span of the protocol. It's necessary to look at the different methods and how they interact with the Signal Sets:

Figure 28: Prepare Inferiors

Parameter	Signals Sets used
Null	All enrolled inferiors are added to the CohesionPrepare Signal Set, this is done as demonstrated earlier by creating a bridge class which sits between the Cohesion and the Atom. Prepare is issued to the Inferior (via the bridge) which in turn calls the Prepare method on the Atom. The result of this is a Vector containing the statuses of all enrolled Inferiors.
Inferior List	The same principle is applied as above however only Inferiors identified in the Inferiors list are added to the CohesionPrepare Signal Set

Figure 29: Confirm Transaction

Parameter	Signal Sets used
Null	All enrolled inferiors must be either confirmed or cancelled. If an inferior(s) has not been issued Prepare then they are added to the CohesionCancel Signal Set, which calls cancel on the Atoms(via the bridge).

	If an inferior(s) have already been issued Prepare and have returned Prepared then these inferiors can be confirmed, hence are added to the CohesionConfirm Signal Set and the work confirmed. If an inferior(s) have been issued Prepare and returned Cancelled then nothing will be sent to the Inferior.
Inferior List	If the confirm set is present (the parameter is not null) and there are inferiors that have not been issued with the Prepare signal then first they must be prepared and the outcome obtained. These inferiors are added to the CohesionPrepare Signal Set which calls the Prepare method on each Atom (via the bridge). If the outcome is success and all inferiors return Success from the first stage then all inferiors identified in this list are added to the CohesionConfirm Signal Set. The CohesionConfirm Signal Set initiates the Confirm method on each of the enrolled inferiors. Otherwise all Inferiors are added to the CohesionCancel Signal Set and all are issued with the Cancel signal. The inferiors that are not identified in the confirm set are added to the CohesionCancel Signal Set which initiates the Cancel method on the remaining inferiors.

Figure 30: Cancel Inferiors

Parameter	Signal Sets used
Null	All Inferiors are added to the CohesionCancel Signal Set, this issues the cancel signal to the bridge which in turn invokes Cancel to the Atom
Inferior List	The same principles as above but only inferiors identified in the list are added to the CohesionCancel Signal Set.

4.2.4.3 – Signal Sets can only be used once

There is also another fundamental problem that needed to be overcome to enable the smooth running of the Cohesion Signal Sets. When a Signal Set is completed and returns its final outcome to the Activity Coordinator the Signal Set is spent and hence can not be referenced again, nor can a new instance of the same Signal Set be registered with the Activity Coordinator. The Cohesion may need to call methods such as Prepare Inferiors and Cancel Inferiors multiple times and therefore make use of the CohesionPrepare and CohesionCancel Signal Sets multiple times. This causes a problem because the implementation of the Activity Service will not allow this to happen, so the Signal Sets need to be registered once with the coordinator when the Cohesion is created and effectively reset under the covers.

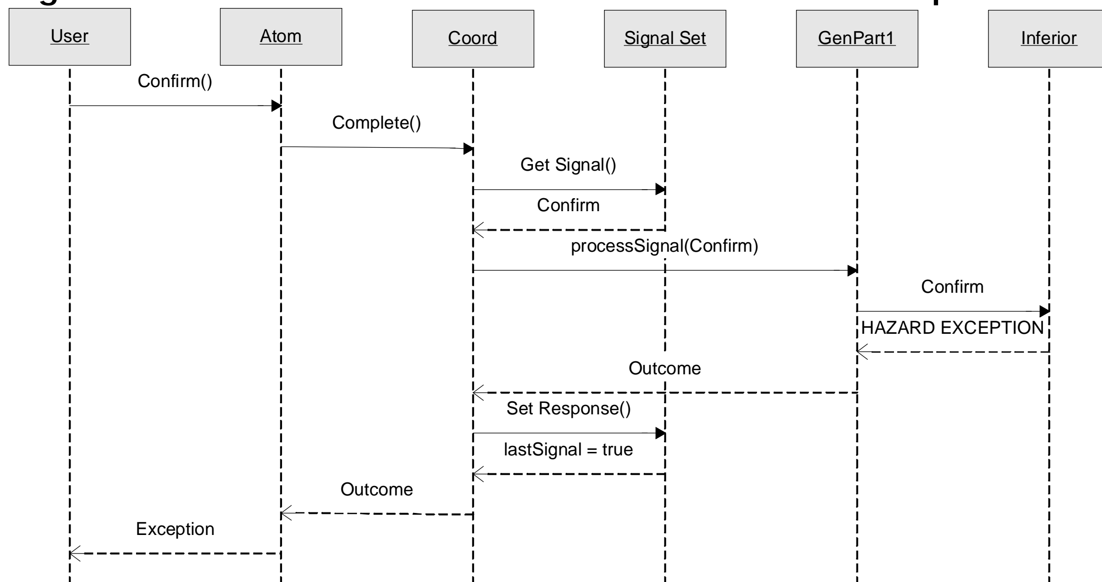
These details are contained within the CohesionPrepare and CohesionCancel signal. When the final outcome is obtained from the Signal Set a variable called ***needToReset*** is set to true. When the Signal Set is referenced again (e.g. Prepare Inferiors needs to be called again) by the get_signal() method, a check is made. If ***needToReset*** is true then the internal variables which indicate that the Signal Set is spent are reset under the covers, so the Signal Set can be returned to its initial state, as if no reference has been made before. This allows the Signal Set to be used multiple times.

4.2.5 - How Errors are handled using the Activity Service

Errors can be caused at any phase of the protocol. These can be Activity Service specific errors, Communication errors, CORBA errors, Errors thrown from an Atom, Cohesion or an Inferior. The important thing is that all errors are masked to look like BTP Exceptions, so that the user of the API only gets useful information back which is specifically related to BTP.

So how does an error thrown by an Inferior propagate up the transaction tree so the user can view the error? Shown below is the series of steps that take place if a User calls Confirm on an Atom but one of the Atoms enrolled Inferiors throws an Exception and for some reason cannot confirm its work.

Figure 31: How errors are detected and thrown as Exceptions



In this example, once the Inferior has received the Confirm signal it throws a Hazard Exception. This Exception is caught by the Generic Participant class, and the error type and the error message are added to the outcome which is sent in response to the Signal.

The outcome is then passed back to the Signal Set in the ***set response()*** method. Inside the Signal Set the outcome data is extracted, if the error type field is not null (in this case it will be Hazard) then the Atom will throw the relevant Exception based on the Error Type field, the propagated message is inserted inside the Exception. The user will now receive the relevant Exception and can decide what steps to take next.

A similar scenario is used if an Exception is thrown while processing a Cohesion, the exception will caught in one of the Cohesions Signal Sets and embedded into an Outcome which will get passed up to the Cohesion and thrown as an exception for the user.

4.3 - Example using the BTP API

To demonstrate the work completed by this project a real life example will be illustrated. Take the example presented in chapter two, a user wants to organise a night out, this involves booking a taxi to the theatre, booking the seat in the theatre and booking somewhere to sleep in a hotel. The user wants all of this work to be done as a single atomic unit, either all of it is booked or none of it is booked. If this transaction cannot be completed then the user wants to order a pizza instead. How can this behaviour be implemented using the BTP API that this project has developed?

4.3.1 – The back end

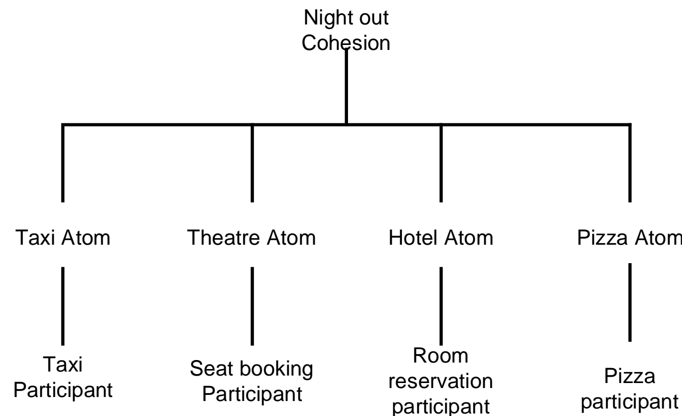
Firstly the services; the taxi, theatre, hotel and pizza shop need to be exposed to the user. Once a reference is obtained to the service, a remote method can be invoked which calls the business logic of the service, if the service can accommodate the user then a participant is enrolled within the context of the transaction. Shown below is a Taxi service, in the constructor of the service a Cohesion is passed as a parameter, this allows the context to be propagated from one remote machine to another and allows the service to invoke methods on the Cohesion created by the user. The service contains a remote method, which can be called. Inside this remote method is the business logic that is provided by the service. If this business logic is successful the method creates a new Atom of work, this requires a String to be added in the constructor, which will be the Atoms identifier when referenced later. Next a new Taxi Participant is created; the participant will be referenced when the commit protocol is initiated. The participant is enrolled with the Atom. Finally the Atom that the service has created (which has one enrolled Inferior) is itself enrolled with the Cohesion. The service also contains a method get_Atom() which returns the Atom of work that is created by the service.

```
public class TaxiService
{
    public TaxiService(Cohesion co)
    {
        _cohesion = co;
    }

    public void bookTaxi(TaxiInformation i)
    {
        System.out.println("Booking Taxi");
        /*
        Business Logic
        */
        try
        {
            //Create a new Atom
            _atom = new Atom("24-7 Taxis");
            //Create a new participant
            TaxiParticipant taxi = new TaxiParticipant();
            //Enrol the taxi participant with the Atom
            _atom.enrol(taxi, null);
            //Enrol the Atom with the Cohesion
            _cohesion.enrol(_atom);
        }
        catch (GeneralException e)
        {
            e.printStackTrace();
        }
        catch (WrongState wrongState)
        {
            wrongState.printStackTrace();
        }
        catch (InvalidInferior invalidInferior)
        {
            invalidInferior.printStackTrace();
        }
        catch (Hazard hazard)
        {
            hazard.printStackTrace();
        }
        catch (InvalidSuperior invalidSuperior)
        {
            invalidSuperior.printStackTrace();
        }
        catch (DuplicateInferior duplicateInferior)
        {
            duplicateInferior.printStackTrace();
        }
    }
    public Atom getAtom()
    {
        return _atom;
    }

    private Atom _atom;
    private Cohesion _cohesion;
}
```

The other three Services look almost identical, the Theatre service creates an Atom, which has a participant that reserves the seat, and the Hotel service creates an Atom, which has a participant that reserves the hotel room. Finally the Pizza service creates an Atom, which has an enrolled participant which books the pizza of the users choice.

The figure shown below is an illustration of the overall services and participants that are taking part in the Cohesion created by the user in this example.

Figure 32: Night out or night in that is the question!



Shown below is an example of a participant that implements the Inferior interface and extends the Asynchronous behaviour, this is the taxi participant. Each participant will be different and will have application specific work included in the prepare, confirm and cancel methods. An ID is also required; this is its unique reference. The method bodies included in this example simply print out messages; in a real life example they would sit in front of a resource and update databases, files etc. in order to carry out work on behalf of the Service.

```
public class TaxiParticipant extends Asynchronous implements Inferior
{
    public boolean prepare() throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed
    {
        //What work to do when Prepare is called
        System.out.println("Taxi received Prepare");
        return true;
    }
    public void confirm() throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed
    {
        //What work to do when Confirm is called
        System.out.println("Taxi received Confirm");
    }
    public void cancel() throws GeneralException, InvalidInferior, WrongState, Hazard,
Mixed
    {
        //What work to do when Cancel is called
        System.out.println("Taxi received Cancel");
    }
    public String get_id()
    {
        return new String("Taxi Participant");
    }
}
```

4.3.2 – The users view

The two code samples shown above are the 'back end' services and participants, so what does a user see? The code sample shown below is an example of how a user (the person who wants to book the Services) would use the API. Firstly a new Cohesion is initialised, and the Service objects are created with the Cohesion as a parameter (in a

normal situation, these would be remote services and a reference to them would have to be obtained). The business logic is then initiated on each of the services by calling the remote methods, passing parameters representing some restrictions to the methods. As a result of the remote method call the business logic is invoked on the service and the application specific participants are enrolled by calling enrol on the Cohesion that was passed as a parameter. The user now decides to prepare all enrolled Inferiors; by calling the prepare inferiors method with a null parameter. The prepare method is called on all Inferiors and is propagated down the transaction tree

```
public class nightOut
{
   public static void main(String[] args) {

        try
        {
            //Create a new Cohesion
            Cohesion c = new Cohesion();

            //Create the Objects
            Taxi taxi = new Taxi(c);
            Theatre theatre = new Theatre(c);
            Hotel hotel = new Hotel(c);
            PizzaShop pizza = new PizzaShop(c);

            //invoke business logic
            taxi.bookTaxi(7.00);
            theatre.bookTheatre("Grease");
            hotel.bookRoom(1);
            pizza.bookPizza("Special");

            //Prepare All Inferiors
            Vector status = c.Prepare_Inferiors(null);
```

After calling the prepare inferiors method with a null parameter the following results are obtained. These results are contained with the status vector, demonstrated above:

Atom	Inferior
Taxi Atom: **Prepared**	Taxi booking participant: **Prepared**
Theatre Atom: **Cancelled**	Theatre seat participant: **Cancelled**
Hotel Atom: **Prepared**	Room booking participant: **Prepared**
Pizza Atom: **Prepared**	Pizza participant: **Prepared**

Because the Theatre atom replied Cancelled to the prepare stage, the user decides not to order the Taxi or book the hotel room because the Theatre is no longer available, the user decides instead to order the pizza, and have a night in. The Taxi and the Theatre need to be cancelled for the transaction to make forward progress.

```
            //Obtain the ID’s of the Atoms
            String taxiId = new String(taxi.getAtom().get_id());
            String hotelId = new String(hotel.getAtom().get_id());
            String pizzaId = new String(pizza.getAtom().get)id());

            //Add the Id’s to a Vector
            status.removeAllElements();
            status.add(taxiId);
            status.add(hotelId);

            //Cancel the Taxi and the Hotel
            c.Cancel_Inferiors(status);
```

Now that the night out has been cancelled the pizza needs to be Confirmed, this is demonstrated in the code sample below:

```
            //Remove all elements in the Vector
            status.removeAllElements();
            //Add the id of the Pizza Atom into the Vector
            status.add(pizzaId);
            //Confirm the Transaction, with the Vector as a param
            c.Confirm_Transaction(status);

            catch (GeneralException e)
            {
                e.printStackTrace();
            }
            catch (WrongState wrongState)
            {
                wrongState.printStackTrace();
            }
            catch (InvalidInferior invalidInferior)
            {
                invalidInferior.printStackTrace();
            }
            catch (Hazard hazard)
            {
                hazard.printStackTrace();
            }
            catch (InvalidDecider invalidDecider)
            {
                invalidDecider.printStackTrace();
            }
            catch (UnknownTransaction unknownTransaction)
            {
                unknownTransaction.printStackTrace();
            }
            catch (Mixed mixed)
            {
                mixed.printStackTrace();
            }
        }
}
```

Even though the Theatre booking participant replied Cancelled to the first stage of the commit protocol the transaction continued to make forward progress, by cancelling the taxi and the hotel the pizza could be confirmed instead.

CHAPTER FIVE

CONCLUDING REMARKS

5.1 - Comments

The original aim of this project was to identify whether or not the CORBA Activity Service was a sufficient coordination model to be able to handle a specification as complex as BTP. To provide a 'proof of concept' implementation of the protocol, and to increase the authors knowledge of transaction and extended transaction concepts. The project has proved very successful and all of the important and interesting concepts of the BTP specification have been implemented using the functionality of the Activity Service. A prototype implementation has been produced and the implementation has been tested by mapping real life scenarios onto the implementation.

5.1.1 –The Activity Service, a useful tool?

What the Activity Service can do for a user has been laid down in Section 3.1, but in practise is this enough, has it provided suitable functionality to support the complex coordination model provided by BTP?

The author was the first person to thoroughly test out the implementation of the Activity Service provided by HP Arjuna Labs, smaller tests were carried out on the implementation but nothing of this scale had been attempted before. The Activity Service was a very useful generic tool that allowed work to be concentrated on the details that actually mattered, (the coordination protocol being implemented) and to be abstracted away from complications of how components should interact. The Activity Service lays down functions and rules which state how coordination is achieved, it is then up to the developer to make use of this functionality to achieve the desired goal.

By providing a simple, relatively 'dumb' Activity Coordinator object that never changes the user only has to be concerned about the logic contained within the Signal Set, and the integration of these Signals with Actions. A different coordinator does not have to be written each time a new protocol is implemented, this saved a lot of time and allowed focus to be placed on the underlying coordination logic.

The author found the Activity Service a rich enough tool to be useful in the development of a complex coordination protocol, BTP. Once the fundamental functions

and concepts of Activities, Actions and Signal Sets were understood then placing BTP on top was still a complex task but without the Activity service would have been even more so. Placing another protocol on top of the Activity Service could be done relatively quickly because the complexities of understanding the Activity Service have already been over come. A user who became an expert at using this tool could effectively and relatively quickly implement another protocol. The Activity Service was very intuitive, logical and complex models could be built up relatively quickly with a skilled user. The most challenging parts of the project were abstracting the details of the underlying Activity Service away from the user and providing the asynchronous behaviour needed in BTP.

After extensive use of the Activity Service the author has a few possible suggested enhancements, these will now be discussed:

remove_signal_set(): The remove Signal Set method only allows you to remove a Signal Set from the Activity coordinator once the Activity that is associated with Activity Coordinator is completed. This makes re-use of Signal Sets difficult, as you have to effectively 'reset' the Signal Set under the covers. It would be very useful if a Signal Set could be removed from the coordinator when it has returned its final outcome and hence has terminated. Then a new instance of the Signal Set could be added to the coordinator and assigned to Actions.

Carrier Protocol: The Activity Service has predefined IDL interfaces and CORBA is used as the carrier protocol. CORBA is a highly flexible and well-understood protocol but with the emerging move towards web services possibly another option of XML encoded messages communicated via SOAP would be a useful addition to this tool and even interoperation between the two.

5.1.2 – BTP, the way of the future?

Traditional Atomic transactions are very useful in tightly coupled systems, which are owned by one organisation. However interactions between autonomous systems that interact on the business-to-business space, where the resources are not owned by one organisation are becoming more and more commonplace. With the drive towards web services BTP is one attempt at providing standardised mechanism for the coordination of resources owned by multiple parties over long periods of time.

After providing an implementation of BTP the author can conclude that BTP offers a more realistic view of transactions over the Internet. It offers interesting features that are

not offered by traditional Atomic transactions. The ability to begin a transaction enrol some participants, confirm some and cancel others enables a user to have an element of choice over the work that is taking place without exclusively locking the resources. This presents a more realistic, real life set of functionality. BTP offers subtle control of long running transactional resources and is an efficient, flexible model. Atoms and Cohesions provide different semantics for a user, enabling traditional atomic transactions to be emulated in a long running transaction, or introducing a finer level of control with Cohesions.

5.2 – Future Work

This project was essentially a proof of concept implementation on the Activity Service, however there is further work that could be done to build on the foundations that this project has laid. Some ideas to build on this projects work are discussed in this section:

Running the implementation over a Distributed environment: The implementation of BTP developed throughout this project simulates the protocol well, but to truly test it, it needs to be run across a distributed environment. The Activity Service runs with CORBA as the carrier protocol, so the interfaces which were defined would need to be run through a CORBA IDL Compiler, to generate stubs and skeletons. This would allow the interfaces to be called from a remote location. Once this programming is done the implementation of BTP could be used in real life scenarios with the Services and the different BTP nodes residing on physically remote machines. The Cohesion, Atom, Service and Service participants would all need to be accessible from a remote location, and hence their interfaces would need to be defined using this IDL Language and the function bodies filled in with the implementation defined by the project. The different BTP roles could then reside on different machines and BTP could be run in a truly distributed environment.

Integrating BTP aware environments: Interactions between BTP running on the Activity Service and that used with XML and SOAP could be very useful. Such that the two BTP aware environments could interoperate to form and end-to-end solution. So BTP in the world of a CORBA developer could interact with a BTP aware participant offered via a web service. This would involve some form of message translation mechanism, enabling the current CORBA calls to be translated to XML.

Exact conformance of the Specification: Most of the BTP specification has been implemented, all of the interesting concepts of the BTP specification have been covered and placed on the Activity Service model. However it is not an exact conformance to the specification so may not correctly interoperate with other BTP aware environments, some extra work needs to be done to make sure that line for line the specification is met.

Qualifiers: The BTP defines a number of standard qualifiers that can be inserted into messages passed between the roles in BTP. Qualifiers can be inserted into certain messages which provide extra information. However the behaviour of the qualifiers has not been implemented, (for example the Prepared timeout value). Some extra interesting functionality could be placed in the implementation of BTP if qualifiers were added to certain messages defined in the specification. The standard qualifiers Transaction Time Limit, Inferior Timeout and Minimum Inferior Timeout would be a starting point.

REFERENCES

[Bernstein97] Phillip A.Bernstein, Eric Newcomer, Principles of transaction processing, for the Systems professional, Morgan Kaufmann Publishers, 1997.

[Leymann00] Frank Leymann, Dieter Roller, Proudction Workflow concepts and techniques, Prentice Hall, 2000.

[OASIS2002] Various authors, Business Transaction Protocol Specification, version 1.0, OASIS committee, June 2002

[OASIS2002] Various authors, Business Transaction protocol Primer, version 1.0, OASIS committee, June 2002

[OMG 00] Various authors, Additional structuring mechanisms for the OTS specification, OMG document number orbos/2000-06-19

[Shrivastava01] S.K.Shirvastava, I.Houston, M.C.Little, I.Robinson, S.M.Wheater, The CORBA Activity Service framework for supporting extended transactions, University of Newcastle upon Tyne, 2001

URL's

OASIS Business Transaction Protocol and Primer

http://www.oasis-open.org/committees/business-transactions/

Activity Service Specification

Object management group

http://www.omg.org/

Activity Service specification

http://cgi.omg.org/cgi-bin/doc?orbos/2000-06-19

Java transaction API

http://java.sun.com/products/jta/index.html

APPENDIX ONE

2PC ON THE ACTIVITY SERVICE

APPENDIX TWO

SOURCE CODE CD

APPENDIX THREE

INTERFACES TO BTP

Inferior Interface

```
public interface Inferior
{
    public boolean prepare() throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed;

    public void confirm() throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed;

    public void cancel()  throws GeneralException, InvalidInferior, WrongState,
Hazard, Mixed;

    public void setSuperior(GenericParticipant gen);

    public String get_id();
}
```

Superior Interface

```
public interface Superior
{
    public void enrol(Inferior inf, Qualifier qual) throws GeneralException,
InvalidSuperior, DuplicateInferior, WrongState;
    public void resign(String serviceId) throws GeneralException, InvalidSuperior,
InvalidInferior, WrongState;
    public Vector request_inferior_statuses() throws StatusRefused,
UnknownTransaction;
}
```

Cohesion Interface

```
public interface Cohesion_Inferior
{
    public Vector Prepare_Inferiors(Vector inferiors) throws GeneralException,
InvalidDecider, UnknownTransaction, InvalidInferior, WrongState, Hazard, Mixed;
    public void Confirm_Transaction(Vector confirm_set) throws GeneralException,
InvalidDecider, UnknownTransaction, InvalidInferior, WrongState, Hazard, Mixed;
    public void Cancel_Transaction() throws GeneralException, InvalidDecider,
InvalidInferior, UnknownTransaction, WrongState, Hazard, Mixed;
    public void Request_Inferior_Statuses() throws GeneralException, StatusRefused,
UnknownTransaction;
    public void Cancel_Inferiors(Vector cancel_set) throws GeneralException,
InvalidDecider, UnknownTransaction, InvalidInferior, WrongState, Hazard, Mixed;
}
```