A framework for the dynamic coordination of services

By
Ian Lawrence

A dissertation presented to the
Faculty of Information Technology
Queensland University of Technology
in fulfilment of the requirements for the degree of
Master of Information Technology

10th July 2007
The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person, except where due reference is made.

Signed: _______________________________

Date: ________________________________
Keywords

Web services, generative communications, service aggregation
Acknowledgements

I would like to acknowledge the patience and support of Stephen Milliner in the completion of this thesis. This thesis represents not only a body of research but also a personal transition from a practical mindset to a concise way of thinking and writing. On many occasions while I saw only trees, Stephen saw the forest and for his enthusiasm and support (beyond the call of duty) I am very grateful.

I would also like to thank David Edmond and Lindsay Bradford for feedback.
Abstract

Web services is a relatively recent initiative that aims to promote program-to-program interaction across the Internet, but while web services is based on a set of XML standards, new standards continue to emerge and existing standards to evolve. Also, web services relies on Remote Procedure Call (RPC) for communication and is thus influenced by the semantics of RPC. In this research, we investigated the juxtaposition of RPC with Generative Communications (GC). GC is a communication paradigm where messages exist independently of the sender and receiver and are stored in a network accessible buffer called a "space": this leads to interactions which are inherently decoupled (in time and space). Also, messages are addressed to recipients by their content, rather than by network addresses, opening up the possibility for one-to-many interactions. These aspects are a marked departure from the RPC paradigm and introduce two main implications: 1) GC messages can be intercepted when in-transit between participants thus introducing the opportunity for mediation and 2) GC can be used as the basis for the aggregation of simple services into more complex ensembles.

In this research, we explored these possibilities by creating proof-of-concept prototypes in three areas. 1) Mediation - GC based mediation was used to intercede between clients and services to allow a client using one protocol to interact with a service using a different protocol. For example, a GC based client interacting with a SOAP service (leading to backward compatibility). 2) Location services - a location service is a GC based service that performs a similar function to a UDDI registry but can be treated as just another service rather than part of an architecture. 3) Aggregation - a workflow design was used as the basis of an aggregated service using GC as the means by which the aggregation elements interact. We concluded that GC provides a natural platform for mediation, location services and aggregation and that these aspects could be combined to produce a holistic service environment.
Contents

1 Introduction ......................................................................................................................... 1
  1.1 The RPC and GC semantics ......................................................................................... 4
  1.2 Impact of GC ............................................................................................................... 5
  1.3 Overview of scenarios and GC based WS communications architecture .......... 8
  1.4 Outcomes .................................................................................................................. 16
  1.5 Research significance ............................................................................................... 17
  1.6 Problem boundaries ................................................................................................. 18
  1.7 Summary .................................................................................................................... 18

2 Literature Review ............................................................................................................. 20
  2.1 Introduction ............................................................................................................... 20
  2.2 Web services background .......................................................................................... 20
  2.3 RPC background ..................................................................................................... 21
    2.3.1 Summary ............................................................................................................ 23
  2.4 Generative communications (GC) .......................................................................... 23
  2.5 The GC-based environment - GC interaction semantics ..................................... 24
    2.5.1 GC coupling semantics .................................................................................... 26
    2.5.2 Summary of GC semantics ............................................................................. 28
  2.6 GC and services ....................................................................................................... 29
  2.7 Applicability of GC to service delivery .................................................................. 31
    2.7.1 Mediation .......................................................................................................... 32
    2.7.2 Heterogeneity Resolution ................................................................................. 32
    2.7.3 Aggregation of Services ..................................................................................... 33
    2.7.4 Other potential benefits of GC ........................................................................ 37
  2.8 Implementation Issues .............................................................................................. 41
  2.9 Summary ................................................................................................................... 44

3 Mediation for backward compatibility ............................................................................ 47
  3.1 Mediator architecture ............................................................................................... 51
  3.2 Proof of concept prototype ...................................................................................... 55
    3.2.1 The mediator window ......................................................................................... 55
    3.2.2 Generic client window ....................................................................................... 57
  3.3 Analysis of backward compatibility and mediation overheads ....................... 58
List of Figures

Figure 1 Proposed architecture for a GC-based mediation system ........................................9
Figure 2 Protocol Heterogeneity Resolution ........................................................................11
Figure 3 Operation of location services ..................................................................................13
Figure 4 Aggregation of services ..........................................................................................15
Figure 5 Five components of the RPC calling sequence ......................................................21
Figure 6 Basic workflow design in the YAWL editor ............................................................36
Figure 7 Decoupled nature of entities in a GC-based environment .......................................38
Figure 8 Multicasting of messages .........................................................................................38
Figure 9 Overview of JavaSpaces™ operations ....................................................................43
Figure 10 Heterogeneous client/service interaction using adapters .......................................48
Figure 11 Access to services with legacy protocols ...............................................................49
Figure 12 Mediation Architecture .........................................................................................52
Figure 13 Sequence of interactions .......................................................................................53
Figure 14 Services Mediator window .....................................................................................56
Figure 15 Web Services mediator interaction .........................................................................56
Figure 16 An example of a generic client window ..................................................................57
Figure 17 Heterogenous interaction using adapters ...............................................................58
Figure 18 Providing SOAP to SOAP interaction ...................................................................59
Figure 19 Example code for a SOAP-GC location service ....................................................60
Figure 20 Example service-side adapter ................................................................................62
Figure 21 Overview of GC-based services ............................................................................65
Figure 22 The publish, subscribe and bind paradigm .............................................................66
Figure 23 Comparison of AOS based client/service interactions .........................................68
Figure 24 SOAP client/service interaction ...........................................................................69
Figure 25 Client/service interaction using distributed behaviour ..........................................69
Figure 26 Typical AOS based interaction ..............................................................................71
Figure 27 Interaction between clients, location services and services ....................................73
Figure 28 Simple sequential aggregated service ...................................................................74
Figure 29 Service Aggregation Prototyping console .............................................................75
Figure 30 Example of simple service aggregation .................................................................79
Figure 31 Example scenario for the OR-Join problem ............................................................81
Figure 32 Holiday booking service designed in YAWL .........................................................83
Chapter 1

1 Introduction

Web services aim to promote program-to-program interaction across a network [1]. One program provides a service and another program, or user of another program (the client), accesses the service by sending a request to the service. The service then processes the request and sends back a response to the client. Whereas currently, the most common use of the web involves business-to-consumer (B2C) interaction, the promise of Web Services is about automated business-to-business (B2B) transactions [2] which it is claimed can reduce the cost of doing business and open up new markets [3].

Web Service technology is relatively new, having been introduced approximately five years ago. This is a short period in comparison to other technologies; for example the development of the relational database took approximately twenty years from concept to wide commercial availability [4]. Web services can therefore be considered to be at an early stage of development [5, 6]. This is further evidenced by the fact that Web Service standards are continuing to evolve with new standards emerging to address additional concerns [7].

Initial Web Service standards were the result of decisions made by a consortium of companies including Sun, Microsoft, IBM, HP and many others. These standards were designed from the start to be open, vendor neutral and based on XML. An important goal of the standards was to promote the decoupling of programs at the implementation level. That is, to allow programs (clients and services) written in any language, running on any hardware and under any operating system to interact and share data [8].

Apart from protocol standards, the Web Services platform also adopted certain architectural and communication mechanisms. For example, Web Services relies on Remote Procedure Call (RPC) technology which was first considered in 1976 [9] (and later [10]) and has become the enabling technology of many distributed computing
systems developed since then (eg. DCE, CORBA and DCOM). However, the use of RPC introduces a number of considerations which have flexibility implications when used in situations that require interaction semantics other than simple one-shot request/response type communications. One example of complex interactions is long-running conversations amongst large and changing sets of participants [11]. These side effects that RPC introduces stem from its original intention, which was to provide remote invocation with local semantics [12].

While web services was originally intended to provide decoupling between clients and services, this has occurred mainly at the protocol level by using XML based standards such as WSDL and SOAP. The use of RPC at the connection level means that web services inherits the network address based coupling and synchronous communications aspects inherent in RPC communications. However, RPC is not the sole communications convention and in this thesis we consider the effects of swapping RPC for alternative Generative Communications (GC) based communications. While web services promote decoupling at the implementation level, GC offers decoupling at the communications level and therefore would seem to better promote the decoupling goal. Specifically, we consider the effort involved in plugging in GC and any benefits that accrue and how, if at all, GC supports the notion of a WS environment.

In this thesis we take this RPC to GC juxtaposition and apply it to three key areas by creating proof-of-concept prototypes: 1) Backward compatibility to existing services using mediation to intercede in client/service interactions (Chapter 3). This enables a client using one protocol to interact with a service using another protocol (eg. a GC client interacting with a SOAP service). 2) Finding services by using “location services” which are similar to UDDI registries but, notably, are not part of an architecture and are implemented and utilised in a similar manner to other types of services (Chapter 4 and 3). The aggregation of single purpose services into more complex ensembles using workflow designs as a basis (Chapter 5). Finally, we draw these three aspects together to propose a holistic service environment that supports the creation, location and invocation of services (Chapter 6).
Generative Communications overview

A service delivery environment (e.g. web services) involves participants such as clients and services that interact while executing concurrently. According to Andrews [13], there are three means by which concurrent processes can communicate:

i) Monitors or shared variables
ii) Buffered messages (message oriented middleware)
iii) Synchronous messages (RPC)

Generative Communications (GC) has been proposed [14] as a fourth communication method. GC is most commonly used by “space-based” programming languages where messages are “generated” into a network accessible shared buffer called a “space”. GC has a number of characteristics which differentiate it from other communication mechanisms.

In a GC-based environment, communications are naturally asynchronous since messages are stored in the space until removed. This buffering introduces the potential for temporal decoupling where clients and services do not have to be active or connected to the network at the same time. For example, a client may generate a message into the space and terminate only to reactivate later to collect a result left by a similarly intermittently executing service. GC also adds spatial decoupling by introducing the concept of associative addressing where firstly, senders address messages by setting the values of public attributes in the messages and secondly, where recipients find messages by matching the message attributes using templates. These aspects are discussed in more detail later.

GC-based clients and services operate in a similar manner to peers in a peer-to-peer network where clients and services have the same opportunities to communicate with each other. This peer-to-peer nature allows clients to contact more than one service at a time, and potentially, it allows services to contact clients. In addition to this, because communications are instantiated in a “space” they can be manipulated during the send/receive process and therefore space-based computing provides a natural framework for interceding in communications and promoting various actions
including for example, mediation between previously heterogeneous clients and services.

All these aspects are a marked departure from the current RPC based semantics of the web services publish-subscribe-bind paradigm where clients interact with services in a much more static request/response style. A key focus of the research has been to investigate how this GC-based intercession can be leveraged to provide mediation facilities that support: 1) Adaptation of existing protocols (eg. SOAP) to allow reuse of existing services 2) Interactions between conceptually heterogeneous services such as service location services and 3) Aggregation of basic services into more complex services. All three aspects are investigated in this thesis with examples of implementation that are used as proof-of-concept prototypes.

1.1 The RPC and GC semantics

While the implementation decoupling aims of web services have largely been achieved by XML based standards such as SOAP and WSDL, at the communication level, web services inherits the interaction semantics of its adopted RPC paradigm. The consequences for WS are:

- The request/response style of RPC communications means that most web services do not naturally support asynchronous interactions since the requestor must wait (perhaps indefinitely) for the response.
- The reliance of RPC on network-address based addressing limits web services to one-to-one communications at the connection level. One-to-many (multicast) communications are not naturally supported.
- In order to provide heterogeneity resolution, some form of adaptation of protocols between client and service must be provided. Since RPC is always a connection between one network endpoint and another intercession in communications is not naturally supported.
- Aggregation of web services is possible but an “engine” external to the web services environment must be provided in order to coordinate interactions between aggregation elements. Aggregation is not a natural extension of the RPC communication protocol.
On the other hand, GC associated semantics support aspects of mediation, adaptation, coordination and aggregation. Hence it appears that investigating GC as an alternative to RPC is warranted and may help support and extend the aims of web services which are firstly to promote the publishing, discovery and invocation of services across the Internet [15] and secondly, to promote decoupling of clients from services.

1.2 Impact of GC

More specifically this research investigates the following characteristics/semantics of GC in order to support the service paradigm as a potential communications and coordination medium.

Spatial Decoupling: rather than the static point-to-point addressing of RPC, GC allows name based addressing that decouples clients from the physical address of a service. This spatial decoupling is achieved by associative lookup [16] which allows messages to be addressed and found by the content of the message rather than a known physical address. Hence, the senders (generators) and receivers (consumers) of messages can remain anonymous if they wish to do so (this is not a natural state of affairs in RPC).

Temporal Decoupling: GC also provides for decoupling in time by allowing for the physical instantiation of messages in the space, independent of the sender and receiver. Decoupling senders, receivers and messages in this manner provides an environment that is 1) naturally asynchronous (since the parties are not bound by an ongoing connection) 2) more flexible (since messages may be exposed to more than one recipient) and 3) more robust (since this decoupling allows for communications with recipients that may be temporarily unavailable).

Dynamic Lookup: Associative lookup also means that heterogeneous description and comparison mechanisms can be used. This allows for dynamic variation in description and “lookup” semantics and may be utilised to complement and extend the pre-existing (static) WS methodologies UDDI and WSDL (see scenario 2 later). Any addressing or matching of resources can be plugged-in (eg. backward compatibility to
WSDL and UDDI) hence promoting inclusion of a broader range of “service-tised” entities (the exact specification of semantics and how descriptions should be defined and matched is beyond the scope of this research).

**Location services:** We extend the above to provide a “service-tisation” of description and lookup facilities via the introduction of a location service. A location service is a concept similar to a UDDI registry in that it provides facilities for service providers to publish service descriptions and for service users to find service descriptions. However, location services are not part of the service architecture (as UDDI is for web services) but are just an implementation of a particular kind of service.

**Decoupled execution:** GC messages are naturally decoupled from the sender and the recipient due to the buffering of messages in the space. This buffering promotes **temporal decoupling** that allows messages to be exchanged between entities that are not active at the same time or where the sender and receiver do not even exist at the same time.

**Multicasting:** Because the space is a “shared” memory, several potential consumers of a message can all see that message – that is, unlike point-to-point RPC, GC can support one-to-one, one-to-many or even many-to-many interactions. This would, for example, allow location services to extend the UDDI concept by allowing them to advertise to clients and also to other services (see scenario 2).

**Aggregation and coordination:** GC provides an environment where interactions between entities can be dynamically and concurrently coordinated and this coordination can be used to enable aggregation of multiple services into more complex value-added services (see scenario 3).

**Mediation:** Because there is a physical incarnation of a message in the “space” the message can be easily accessed, transformed or replaced. Hence, GC also provides a natural platform for mediating communications between potentially heterogenous communicating parties. Mediation is a fundamental aspect of the research and is used in all scenarios and presented in more detail in the next section.
These factors support the case that GC is, as Gelernter proposed [14], a separate category of communications, similar perhaps to message passing, but sufficiently different to be considered in its own right. Therefore, in this thesis we investigated how the GC paradigm can be used as an alternate communications infrastructure for service delivery. In particular, we investigated GC as an enabling technology to create a layer of mediation that can be used to intercede between clients and services to intelligently and dynamically handle various aspects of client/service interaction. To demonstrate these aspects we consider three cases or scenarios:

In case 1 we look at how the GC approach can be utilised with the current web service protocols (SOAP) and perhaps extended to include others such as for example CGI and CORBA.

In case 2 we examine how GC-based mediation can be used to intercede naturally between clients and services to replace services with other similar services at runtime. If interface differences exist, these can be resolved during mediation by using adaptation. We also introduce location services as an extension of UDDI registries.

In case 3 we investigate how GC can be used to promote the aggregation of services by providing a means for aggregation elements to communicate and coordinate in an asynchronous and decoupled manner.

In all cases, where applicable, we examine any benefits that can be obtained from the passing of behaviour. For example, Java based GC passes messages as Java objects which can carry not only data but also behaviour in the form of methods. This allows two key benefits; 1) processing can be transferred to the most suitable location (client or service) and 2) service interface details can be encapsulated into executable code which can be passed to the client thereby reducing reliance on hard-coded interfaces.

As a conclusion to this work we postulate a service “environment” which combines all of these aspects into a coherent set of software components that can be used by 1) service developers to publish not only new services but also existing services that use a variety of protocols 2) service users to initially find services and later to invoke and
interact with services and 3) aggregate service developers to design, translate and publish aggregate services.

In Chapter 2, we present a review of current literature relating to services, service paradigms and service environments. In Chapter 3, we investigate the use of GC to re-use existing heterogeneous services such as services supporting SOAP. In Chapter 4, we investigate the ability of GC to support behaviour passing and also introduce service location services (similar to UDDI registries). In Chapter 5, we investigate the use of GC to provide service aggregation facilities. In Chapter 6, we propose an architecture for a service environment which promotes a holistic approach to supporting service creation, publishing and invocation by heterogeneous clients and services. In these chapters, proof-of-concept prototypes are used to support the investigation.

In the remainder of this chapter, a detailed overview of the proposed GC-based architecture is provided and an overview of the prototypes developed in chapters 3, 4 and 5 are presented.

1.3 Overview of scenarios and GC based WS communications architecture

In this section we introduce the architecture used to promote mediation and aggregation in a GC-based environment and present some details of a set of use case scenarios to illustrate key aspects of this architecture.

The high level aims of this architecture are to:

- Provide GC-based mediation services that:
  - Promote interaction between heterogeneous clients and services (adaptation)
  - Facilitate the re-use of existing services
  - Allow creation of new GC-based services
- Leverage distributed behaviour (decoupled execution) to reduce reliance on heterogenous service interfaces
• Use GC-based techniques to facilitate the aggregation of simple services into more complex ensembles
• Promote integration of the aspects just mentioned into a general service environment (mediation, adaptation, aggregation)

![Diagram](image)

Figure 1 Proposed architecture for a GC-based mediation system

The central part of this architecture (Figure 1) is mediation which relies on access to the GC “space” to monitor message movements between clients and services. Messages are intercepted during mediation by matching patterns of attributes contained in the messages. Service developers can either create new GC-based services or they can re-use services from existing protocols (eg. SOAP). Location services would be one of the first types of services to be published as this would allow service developers to also register a service for discovery by clients – examples would be the “service-tisation” of existing WSDL and UDDI services.

The mediation component
The mediator is pre-programmed to create adapters for a particular protocol (eg. SOAP). If a new protocol is to be handled then this ability must be programmed into the mediator. When services are published, the “publish” message is intercepted in the mediation layer and the adapter, or wrapper, for that protocol created for each service.
Chapter 1: Introduction

The adaptor then intercedes in all communications and is transparent to the client and the service. Adapters translate the generic protocol on the client side of the interaction to the specific protocol used by the service (SOAP, CGI, CORBA, GC etc). Clients find services by querying location services that are published the same as other services. When clients interact with services, they do so by communicating with the service adapter using a generic protocol (not the native protocol of the service).

The aggregation components
Aggregate services are created from workflow plans and then translated into a set of GC based services that represent the elements of the workflow. The aggregate service is published and accessed the same as all other services. When the aggregate service is invoked, the elements interact with each other to perform the intention of the workflow (this may involve contacting remote or GC based services).

1.4 Services in the proposed architecture

In this thesis, we refer to several types of services. The term “service” can be applied to any unit-of-computation that is accessible over a network: a service can accept requests, carry out computation and may return a response. We further divide the notion of “Service” into two categories, services that use existing protocols (such as for example SOAP and CGI) and GC-based services (see Appendix 2). GC-based services represent services that were constructed to use GC protocols as their primary means of communication. Services that use existing protocols can be “wrapped” in GC-based code to become GC-based services. Also, two categories of clients are referred to - GC-based clients and other clients such as for example SOAP or CGI based clients. Three use-case scenarios are now presented to highlight how GC-based mediation has been applied in this research. For simplification, we initially assume all clients are GC-based and consider protocol heterogeneity at the service end rather than the client end.

Scenario 1: Protocol Heterogeneity

This scenario promotes backward compatibility to existing services from a variety of platforms (eg. CGI, SOAP etc) by providing a layer of adaptation that converts the
details of the service interface into a common GC-based protocol. A generic invocation interface window (programmed by the service developer) is used by the client to access each service. Therefore clients do not have to write interface level code (such as SOAP etc) to access a service and do not need to be concerned about the specifics of the service protocol.

In order to allow a GC client to connect to services that use different protocols (eg. CGI, SOAP etc) mediation can be used to intercede between the client and service to provide adaptation of the client request to the appropriate protocol. An adaptor is written for each service to carry out the translation from the “legacy” protocol to the generic GC protocol expressed in the adapter logic. For example, a stock quote service may contain a method called getPrice with a single char parameter called code. This method would also be implemented in the adapter as getPrice(char code) but the adapter would translate this call to the protocol of the remote service using any platform specific tools (JAR files or DLLs) required. The return value would be translated back to the generic protocol in a reverse manner. The developer of the client program need only be familiar with the semantics of one protocol (the generic protocol) not a suite of disparate protocols (eg. SOAP, CGI etc).

![Figure 2 Protocol Heterogeneity Resolution](image-url)
As an example, in Figure 2 the mediation layer extracts WSDL information such as service name, method names and their parameter names and types from candidate services that are published (step 1). A GC client then selects a service from the mediation layer by using a location service and receives a copy of the generic invocation interface (step 2). The client then interacts with the service, through the mediation layer, using the generic invocation interface (steps 3 and 4).

Key points from the scenario are:

- Clients can use generic interfaces to access services. This reduces the necessity for clients to understand interface details
- Mediation layer manages interaction between the client and the service
- Remote service developers can make their services available in a protocol transparent manner – so for example, SOAP or CGI services are viewed equivalently by a client.

**Scenario 2: Conceptual Heterogeneity**

*In this scenario, location services are introduced as a concept similar to UDDI registries. In fact, a location service may simply be a “wrapped” UDDI registry thereby promoting backward compatibility. Location services extend the use of distributed behaviour by passing query interfaces and invocation interfaces as executable code.*

UDDI registries are used by web service providers to advertise their services and by web service clients to locate suitable services. Similar to UDDI registries, location services provide registration facilities to potential services and registry-like, service location services, to clients. In addition, because location services are GC-based, they operate more as peers to other services rather than as part of the architecture like UDDI registries. Their peer-like nature means that they can more easily participate in the service environment. For example, location services can advertise their presence to clients and to other location services. Also, in contrast to UDDI registries, location services “pull” requests from clients from the space rather than having the requests pushed to them by clients. This “pull” based messaging means that requests may be
serviced by any location service that is “suitable”. In the scenario presented here, other services, such as for example printer and fax services, are introduced as additional examples of GC-based services.

**Example of scenario 2**

Location services are created and introduced into the mediation environment where they immediately advertise their presence by placing advertisement messages in the space. The location service also places a service description object (SDO) in the space that is used by prospective services to register with the location service. When developers create new services (such as printer or fax services in our example scenario), these new services can be registered with the location services by reading a copy of the SDO, filling in the details of the service and returning the SDO to the space where it is retrieved by the location service. The registering service must include not only a description of the service but also an invocation interface (cf. scenario 1) in the form of a GUI (or other interface component eg. stub code) that can be used by clients to assist the invocation the service. When the location service finds a matching service for the client it returns an object to the client that includes the invocation interface which the client uses to interact with the service.

![Figure 3 Operation of location services](image)

Key points from this scenario:

- Location services are not part of the GC-based service architecture (cf. UDDI registries) but are just an implementation of a service. They can advertise their presence to service providers, clients and other location services.
• Services and clients interact with location services in a similar manner to UDDI registries
• The service provides an invocation interface for the client which reduces the reliance on knowledge of interface details.
• Just-in-time download of the invocation interface leads to simplicity of use of services by clients
• By essentially breaking the publish, subscribe and bind service architecture “triangle” everything becomes a service and is more generic and decoupled.

**Scenario 3: Aggregation of services into a composite service**

*GC can be used as a mechanism to coordinate the interactions between the component tasks of an aggregated service. Coordination logic is distributed among the tasks rather than being managed by an external “engine”.*

In this thesis, aggregation is viewed in terms of a workflow. One approach to workflow management is to provide a centralised “engine” which manages the interaction between all of the elements of the workflow. Some examples of a centralised approach are YAWL and IBM MQSeries/workflow [17]. Another approach is to distribute the interaction logic that directs the workflow, among the elements that make up the workflow. These two approaches have been described as exogenous and endogenous respectively [18].

Using the architecture described earlier in Figure 1 the following scenario shows how a workflow design can be translated into a set of tasks that perform the requirements of the workflow design:

• The tasks are implemented as a set of GC-based Java classes that coordinate the interactions of the workflow.
• Each task may provide internal processes that invoke a service (local or remote).
• Each task contains logic to handle combinations of incoming messages and logic to generate combinations of outgoing messages.
• The client invokes the aggregated service as if it were a single service. That is, underlying implementation of the workflow is transparent and the aggregated service appears as a single service.

Figure 4 Aggregation of services

Sequence of operation
1. Workflow design is created using the YAWL editor (or other workflow design tool)
2. Workflow is translated to a set of GC-based classes to represent the tasks of the workflow. The flow logic is embedded in the task classes.
3. The mediator manages the interaction between the client and the aggregated service
4. The tasks communicate with each other using GC
5. Messages pass from one task to another based on the workflow
6. Tasks that need to contact local or remote services do so as the flow progresses. When the end-point is reached, a response is returned to the client.

Key Points from this scenario:

- A workflow design of an aggregate service is translated to a GC based implementation
- Conditional logic and coordination information is encoded into each task
- The coordination of the flow between tasks is managed by the tasks
- Tasks may utilise either local or remote services

1.5 Outcomes

The goal of this research is to investigate the use of GC as an alternative communication mechanism to RPC and explore how this can be used to create a holistic service delivery environment. Aspects of the use of GC will be verified by creating proof of concept prototypes based on the test scenarios in the previous section. The main objectives are:

- To demonstrate backward compatibility to services written for existing protocols such as for example SOAP and CGI using GC-based mediation (See scenario 1 and Chapter 3).
- To demonstrate the use of location services (an extension of UDDI registries) while still allowing the re-use of existing UDDI registries (See scenario 2 and Chapter 4).
- To demonstrate the viability of GC-based communication and coordination in the creation and execution of aggregated services (See scenario 3 and Chapter 5).
- To demonstrate the usefulness of Java based distributed behaviour by creating invocation interfaces that reduce the need for client programs to have detailed knowledge of service protocols and interfaces (See scenarios 1 and 2).
- Finally, to combine these objectives and propose an architecture or framework for a service delivery environment that can be used to assist service providers, service searchers and service users (See Chapter 6).
1.6 Research significance

Generative communications is a communication mechanism that exhibits two degrees of decoupling not naturally supported by RPC communications; spatial decoupling and temporal decoupling. The significance of this research is to determine the benefit that these attributes bring to a service environment in particular in the provision of mediation services. The main focus of mediation in this research is the intercession in interactions between clients and services and how this can be leveraged to provide access to heterogeneous services and in particular, to provide backward compatibility to existing services that use disparate protocols.

Java based GC introduces the ability to pass behaviour between participants. In web services, WSDL and SOAP are used to decouple clients from services at the implementation level. In this research, behaviour passing is used to further decouple clients and services by passing invocation interfaces that allow services to be invoked without the need for interface level programming specific to the service. The invocation interface is a self-contained program passed as a Java object that can be invoked by the client and used to interact with the service.

As a third aim/goal of this research, we proposed a means of creating aggregated services from simpler services taking into account the full development lifecycle from design to execution. Most aggregated service implementations adopt an exogenous approach where coordination is imposed from outside of the aggregated service. An exogenous approach can lead to programs that perform coordination only, making them easier to understand and easier to reuse [18]. The approach we have adopted is an endogenous one by distributing the coordination logic among the aggregation components and then using GC as the communication medium. This distributed approach to workflow coordination has the advantage that elements of the aggregation can be replaced without affecting other elements. The application of GC to service aggregation in this manner has not to our knowledge been attempted before.
1.7 Problem boundaries

The scope of this project is constrained in the following ways:

- Methods used for the discovery of services are not explored in this research.
  For the purposes of testing, a small set of commercially available web services such as Amazon, Google and NASDAQ are used.
- When considering the passing of behaviour between clients and services, issues of security are important. However, this is a large topic in itself and is not the focus of this research.

1.8 Summary

Current service environments such as web services rely on RPC communications for interactions between clients and services. The original intention of RPC was to provide transparent remote interaction with local semantics, however, the assumption that a RPC is local is naïve with respect to a service environment. The majority of web services are simple one-shot request/response type services that work effectively, but more complex services that involve an aggregation of interactions from a number of services covering an extended time period still present some issues [19]. Since web services is a recent innovation and still in the early stages, it may be prudent at this stage to be investigating and critiquing communication mechanisms other than RPC which offer a set of attributes that are a more natural fit for the provision of service interactions. Such interactions may be simple or complex in nature and short or long-running in duration.

Generative Communications is one such alternative communication mechanism; it provides a set of attributes that, in juxtaposition to RPC, may be more suited to complex, long-running service interactions. GC interactions use associative addressing and are naturally asynchronous due to the inherent buffering of the “space”. Also, Java based GC introduces the ability to pass behaviour between participants. Therefore, GC should be considered as an alternative communication mechanism for the delivery of services and may provide a better basis for a more
holistic approach to services via the development of an encompassing service environment.
Chapter 2

2 Literature Review

2.1 Introduction

In Chapter one, we indicated that GC can be considered as an alternative to RPC. In this chapter we look into the background of each of these in order to then compare and contrast them. The chapter concludes with a review of workflow based service aggregation.

2.2 Web services background

The Web Services (WS) movement is the result of collaboration between several large information technology organizations [1] and other smaller companies. Having only been introduced approximately five years ago WS standards are still evolving, emerging and in some cases competing [7]. While this evolution has led to some maturity for one-shot request/response type services there is still work to be done with service interactions that are complex and long running [19].

It is important to note that developing a new technology is difficult. For instance, WS is still in its early stages [20-22] in comparison with the time taken for the development of the relational database. Following on from Codd’s relational data model in 1970, it took approximately a decade to develop the infrastructure and languages and then another decade before robust commercial relational database management systems were available [4].

Given the relatively early state of web service development and the continuing investigation and development of the many web service standards, it seems prudent at this time to also be investigating, critiquing and providing alternatives to the current architecture choices underlying web services – in the case of this thesis, the adoption of Remote Procedure Call (RPC) communications infrastructure. In the next section RPC is examined more closely.


2.3 **RPC background**

*Definition:* Remote Procedure Call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel [23].

The Web Service paradigm is based on RPC style communications. While RPC programming tries to imitate local programming, the reality is that issues such as network latency and service unavailability create problems when distributed processes rely on each other for timely interaction. In addition to this, a network address must be known before an RPC connection can be made which limits the connectivity of the RPC paradigm to one-to-one concurrent interactions using statically known network addresses [24]. Thus, RPC communications have definite associated semantics which impact on interactions between disparate parties.

In a program running on a single computer, data and control can be transferred from one part of the program to another by making (local) procedure calls. Remote procedure calls are modelled on this principle but have been adapted to allow data and control to be passed from a program running on one computer to a program running on another computer [10]. One description of this process is an “Inter-process communication disguised as a function/procedure call” [25]. The RPC architecture can be portrayed as consisting of five components as shown in Figure 5.

![Figure 5 Five components of the RPC calling sequence](image-url)
Chapter 2: Literature review

Remote Procedure Call requests flow from the client to the server and the response flows back as depicted in Figure 5. The initial intention of RPC was to provide a means of allowing one program to call another remote program in a manner that was largely transparent to the programmer [10, 26]. Some RPC related issues which impact upon the notion of service provision are as follows [12, 25]:

1. **Client and server roles**: Sometimes the roles of client and server need to be reversed, for example: distributed clients that cache data from a central database must be notified that their caches need updating. For RPC communications, this requires that the client take on the role of server but in RPC based implementations the client and server roles are usually predetermined. The ability to perform both client and service roles adds flexibility and dynamism to a service environment.

2. **Hand Shaking**: Given that any communication from one entity to another can be lost (including not only the original message but also the acknowledgment), how can successful communications be guaranteed? Lost messages in an RPC environment are not recoverable from the network. Therefore, reliable communications are important to a robust service environment.

3. **Multicast**: The ability to concurrently send messages to multiple recipients adds flexibility to a service environment. RPC does not naturally support multicast communications because RPC communications are conceptually limited to one-to-one connections between statically known addresses.

4. **Repeated execution semantics**: Some operations are idempotent such as reading values from a file, others are not, such as updating an account. If an RPC fails then repeating a non-idempotent call could cause problems. For reliable operation, a service environment must be able to correctly interpret duplicate requests.

5. **RPC is stateless**: State information must be stored at the client and/or the server. If either crashes between RPCs then state will be lost. In a service environment this would mean that state would have to be managed by the service leading to additional overheads.
2.3.1 Summary

While RPC can be a useful tool for connecting and coordinating remote processes, there are a number of implications that arise due to its original intention – *to make remote calls appear as local calls*. Issues such as one-to-one communications and intolerance of network latency appear as open challenges. Any attempt to turn RPC into a more flexible protocol would require additional layers of supporting software to be added, specifically, with respect to roles, multicast and state etc. Therefore, when considering a communication mechanism to support a service environment perhaps the most beneficial protocol would be one that assumes (a priori) that networks and the heterogenous entities that populate them are dynamically changing and intrinsically somewhat unreliable. The associated semantics of RPC essentially limit its ability to act in these required ways. Generative communications on the other hand has completely different associated semantics and, as discussed below, these go some way toward addressing the concerns raised.

2.4 Generative communications (GC)

The view has been presented by Andrews [13] that concurrent programs communicate in three ways; 1) Monitors (shared variables) 2) Message passing (Message Oriented Middleware) and 3) Remote operations.(eg. RPC). However, GC has been proposed by Gelernter as an additional form of communications [14] that is “sufficiently different” from the three presented by Andrews to warrant its inclusion as a fourth model. Although GC is similar to message passing, there are also a number of significant differences [27].

The concept of GC was originally centred on a “tuple-space” or “space”. A space is a network accessible shared memory and rendezvous-point into which tuples containing data can be stored and from which tuples can be either read or retrieved. The term “generative” is applied to this paradigm because tuples are generated into a tuple-space. One of the distinguishing features of GC as a communication paradigm is that until a tuple is withdrawn from tuple-space, the tuple exists independently of the entity that created it and independently of any entity that may retrieve it. This allows participants to interact in a decoupled and anonymous manner if required. Messages are found and removed from the space by matching a template against visible values
from within each message; this is called associative lookup [16]. Because physical addresses are not used and all messages are buffered in the space all participants are decoupled in time and space. Clients and services in the system are effectively decoupled from each other and this allows participants to be added and removed without breaking any contracts.

Languages that exploit GC usually consist of a set of primitive operations that allow tuples to be written to (in), read from (read) or removed (out) from, a tuple-space. The earliest example of this kind of language is Linda [14]. Traditional communication models such as Remote Procedure Call (RPC) usually rely on the passing of messages between participants in a system where the participants and their locations are known to each other. The participants must also be active at the same time. GC adopts a different model. GC is a system where messages are generated by participants and passed into a shared storage medium (tuple-space) where they may be retrieved by other participants [14].

Ordered tuples are usually passive data but in some environments such as the Java based JavaSpaces [28] or Active Object Space [29] tuples may contain executable code. These “live tuples” [30] are sent to the tuple space, removed by a recipient, execute some process such as gathering some data (independently of the originating process) and then return to the space as passive tuples for collection by the originating process (or any other process for that matter). This introduces the concept of distributed behaviour which can be beneficial in a number of ways; 1) processing can be moved to where it is most effective or to reduce bandwidth consumption and 2) user interfaces can be passed from services to clients thereby eliminating the need for the client to know the service’s interface details. Instead, the client has knowledge of a very simple and generic execution pattern (eg the command pattern [16]). This latter aspect is explored in this research in Chapter 3.

2.5 The GC-based environment - GC interaction semantics

One focus of this research is the development of a framework for a GC-based service delivery environment. In order to achieve this, several key aspects of the GC paradigm are leveraged. One of the most obvious semantic aspects of GC is the
“space” which can be used as a rendezvous point for mediation of service interactions. A mediator can then be written to intercede in communications between clients and services. A mediator, in the context of this research, is an autonomous program that is transparent to clients and services and can monitor and interact with GC messages being passed between clients and services. Since this autonomous mediation and the role GC plays in interaction is important these will be examined in more detail now and in following sections.

Autonomous programs (such as in our case mediators, clients and services) that are designed to communicate with other autonomous programs have been considered previously by Wegner [31] as being superior to Turing Machines [32] which are programs that take an input, make a computation and produce an output. Wegner called these autonomous programs “Interaction Machines”. One use of GC environments is to provide the medium through which interaction machines such as mediators communicate.

Traditionally, computer systems have been created from components that are algorithmic in nature. That is, a component performs a calculation based on some input, produces some output and then expires. This behaviour was first described by Alan Turing [32] and these “units of computation” have become known as Turing machines. Wegner has described the observability of this behaviour as: *computable functions which transform inputs into outputs* [33].

Wegner goes on to define a concept called an “Interaction Machine” as an object which extends a Turing machine by adding the ability to add input actions from outside of the function while computation is taking place (Turing machines are closed to the outside world during computation). Wegner contends that this simple addition changes Turing machines from closed systems into open systems with far richer behaviour than Turing machines [31]. While the observable behaviour of Turing machines can be expressed as a transformation of inputs into outputs the observable behaviour of interaction machines cannot be expressed by transformations since the observable behaviour of interaction machines is inherently unpredictable in terms of the time taken for interactions and the sequence of events that occurs. A mediator is an example of an interaction machine because on the one hand, it is an example of a
Chapter 2: Literature review

Turing “unit of computation” and on the other hand, using GC it can receive and send communications while it is computing (cf. intercession). The additional possibilities of interaction that interaction machines bring, as extensions of Turing machines, adds to their importance as a computing concept [18].

It would be possible to have Interaction Machines communicate using RPCs but this would create an environment where each Interaction Machine would need to know the network address of each collaborating interaction machine either at design time or it would need to be obtained from a central registry. GC semantics on the other hand, naturally support a flexible means of communication for interaction machines and allow interaction machines to address collaborators in a more general manner. For example, using GC it would be possible for interaction machines to address each other by name rather than by network addresses. This opens up the possibility for one-to-many interactions. In this research, mediators are the Interaction Machines and GC provides the communication channel through which interaction takes place.

Simple request/response style services that are similar to Turing machines have limited scope for dynamic service interactions. On the other hand, services that have similar attributes to Interaction Machines have the ability to interact with other services rather than just a simple request (input) and response (output). Thus, interaction machine style services could promote a service environment that is more flexible and dynamic. For example, an interaction machine style service could receive a number of inputs concurrently, carryout processing and then create multiple outputs (concurrently). Inputs and outputs could even be occurring while processing is taking place. This is in sharp contrast to the simple input-process-output model of the Turing machine style service.

2.5.1 GC coupling semantics

One means of allowing autonomous programs or objects to interact with each other is the tuple-space concept. Recapping, a tuple-space is a network accessible shared repository (sometimes called a blackboard, board or space) which is capable of storing entities called tuples. Tuples can be written to, read from and taken from the space.
during the interactions between objects. These primitive operations are the basis of space-based computing.

GC-based computing introduces the concept of systems that exhibit spatial and temporal decoupling [34]. That is, systems can be created that are; 1) distributed across a network (spatial decoupling) and 2) communicate with each other in a time-decoupled manner (temporal decoupling) due to the buffering that is naturally exhibited by the tuple-space. A more precise definition of temporal decoupling in this context, however, is: Communication between entities that are not necessarily active at the same time [35] or entities that are part of a dynamic system that may not have been created yet.

The importance of spatial and temporal decoupling in building a dynamically interactive system is as follows:

- Computation can be distributed among the components that are interacting thus avoiding the need for a centralised coordinator. This can lead to scalability and increased performance if the components are located on separate machines [36].
- New interaction components can be created either at run-time either before the coordination process starts or during the coordination process.
- New components can be created on-demand based on decisions made by other components or in response to real-time events such as time-outs related to the unavailability of services.

The attributes of spatial and temporal decoupling would appear to support a flexible and dynamic form of service delivery.

**Relevant Literature Summary**

- Background information about space based environments and generative communications can be gained from the early work by David Gelernter [14] and later [37]
• Information about more recent space based systems can be obtained from Freeman, Hupfer and Arnold’s in-depth examination of the JavaSpaces™ system available from Sun Microsystems [16].

• A broad review of coordination across all disciplines can be obtained from Malone et al [38].

• A paper describing a system of workflow coordination can be obtained from Benatallah et al [36]. The system proposed by this paper is very similar to my initial concept with the exception that I propose a space-based approach to solve the same problem.

• Mediation of services to resolve heterogeneity issues is treated in Nagano [39] and Benatallah et al [36]

### 2.5.2 Summary of GC semantics

Many currently accepted concurrent programming paradigms use communication techniques that involve messages passed either via shared variables in the case of a standalone program or by RPC style point-to-point communications in the case of distributed systems. Both of these systems create a tight coupling between the participants which can lead to a lack of flexibility.

GC is a promising alternative to RPC based distributed computing paradigms such as Web Services, CORBA, Java RMI and others. GC allows concurrent processes to communicate in a decoupled manner promoting dynamic and flexible collaboration and coordination. A number of GC based development environments are available including Active Object Space, which has been chosen as the basis of this research.

The nature of RPC means that only one service can be addressed at a time and the address of the service must be known before the call is made. This static addressing not only precludes the use of multicasting, where more than one service can be reached with a single call, but also limits the use of dynamic addressing where the address of the target service can be changed at run time.

In comparison, space-based programs using GC provide an environment where participants can interact with each other anonymously if required. Communications
can be directed to one, many or any recipients. Messages take the form of tuples, which are generated into a “tuple space” where they can be read or removed by any interested parties. Various incarnations of this paradigm have been developed such as Linda, T-Space, JavaSpaces and Active Object Space. The decoupled nature of GC, in both time and space, creates an environment where clients and services can be connected together more dynamically than using RPC style statically addressed, request/response style communications. The decoupled environment that GC provides is suitable for the creation of a layer of mediation which can intercede between clients and services to add additional capabilities such as:

- Heterogeneity resolution
- Capacity management and monitoring
- Availability management
- Legacy Systems
- Aggregation of services

In the following sections we will compare RPC to GC more closely and investigate mediation in greater detail.

### 2.6 GC and services

While perhaps the most prominent similarity between RPC and GC is that they are both examples of communication paradigms the differences between them are numerous because of their different interaction semantics.

**Decoupling**

While the aims of web services are to provide decoupling at the implementation level [8], the main aim of the RPC communication mechanism on which web services rely is to provide remote calls with local semantics. Web services therefore, inherits these semantics, that is, static addressing and synchronous communications. GC on the other hand allows 1) spatial decoupling by replacing network addressing with associative addressing and 2) temporal decoupling by using buffered communications.
Chapter 2: Literature review

Addressing
RPC communications rely on network addresses for communication between clients and services and this introduces a number of implications. Firstly, the address of any potential service must be known in advance of communication being established and secondly communications are one-to-one or unicast. On the other hand GC does not have the notion of communication between defined end-points and the location of the receiver is unknown to the sender. This introduces the potential for multicast communications.

Asynchronous communications
The most common pattern of use for web services is the simple request/response style of communication. In this mode of operation, the client sends a request to the services and waits (i.e. synchronous) until a response is received. While this may suffice for simple interactions, more complex long running interactions could suffer if network latency or outages occur. GC however, provides a natural buffering for all messages as they rendezvous in the space. This inherent asynchrony means that by default, GC-based programming languages must handle asynchrony as a matter of course and that GC programmers must take asynchronous communications into account at all times.

Intercession
The ability to intercede in communications between clients and services introduces the opportunity to modify or redirect messages transparently to both the sender and receiver. Since RPC semantics require communications to be carried out between network addresses the opportunity for intercession is not available. On the other hand, the buffering and rendezvous of messages provided by GC allows for a natural means to intercede in the message passing process.

In section 2.3, we listed a number of issues that are relevant to RPC communications. These are now explored with respect to GC.

1. **Client and server roles**: In a GC environment there are no clear client and server semantics. Each GC participant is capable of fulfilling the role of either a client or a server depending upon how it is configured. GC-based participants operate more as peers in a peer-to-peer interaction
2. **Handshaking:** Because GC messages (and responses) are buffered in the space, the potential for lost messages due to problems with senders or receivers is reduced. A message sent to an off-line recipient will be stored until the recipient is on-line. Messages which accumulate in the space, can be removed by a “lease” concept that is available in some implementations of GC.

3. **Multicast:** The associative addressing aspect of GC naturally supports multicasting of messages. Associative addressing allows a GC participant to address messages to other GC participants by setting public attributes in the messages. In this way, a message may be addressed to a discrete recipient or to a class of recipients.

4. **Global variables:** GC allows the “space” to be used as a repository for messages that can be accessed by any or all GC participants. Simple variables or complex structures such as linked lists can be stored in a space.

5. **Repeated execution semantics:** Since repeated requests for a service would result in multiple messages, any superfluous messages would remain in the space until removed. This would allow the service to detect that multiple requests had been received and to deal with the additional messages appropriately.

6. **State capture:** In a GC environment messages exist independently of senders and receivers while they are being stored in the space. Therefore the state of any GC participant can be stored in a message.

In this section, we have been looking at the juxtaposition of RPC and GC as underlying mechanisms of interaction between clients and services. In the next section, we shift the focus to a higher level and examine the use of GC as a means of providing middleware-style facilities to: further decouple clients from services (mediation and heterogeneity resolution) and to create more complex services from basic services (aggregation).

### 2.7 Applicability of GC to service delivery

Previously, we have been looking at RPC in comparison to GC from a communications protocol perspective. In this section, we examine some specific
issues to which GC can be applied in the provision of a service delivery environment. These include mediation, heterogeneity resolution and service aggregation.

2.7.1 Mediation

Mediation as defined in this research is the intercession in communications between senders and receivers. Mediation provides the opportunity to intercept, modify and redirect messages between senders and receivers transparently (that is, without the knowledge of either the sender or receiver) and to monitor message traffic through the service environment. Since RPC based communications semantics involve a connection from one endpoint to another at the network address level, mediation is not naturally supported. GC-based mediation can be used to address several aspects of services and these are now discussed in more detail.

2.7.2 Heterogeneity Resolution

Services may need to be substituted for other similar services at run time due to reasons such as network latency, denial of service (eg. flash crowds) and routine service outages. When substitution is required, any differences between heterogeneous service interfaces will have to be resolved. GC based mediation can be used to intercede between clients and services to adapt the client request to the appropriate service interface. Heterogeneity however, can be viewed at different levels.

Semantic Level

Semantic heterogeneity exists when the same data or operations are found in multiple services under different names. This is similar to semantic heterogeneity in databases [40]. One use for mediation would be to substitute services at run time in response to service outages in order to resolve semantic heterogeneity. However, even services that carry out the same task are most likely to have different names for parameters and operations. Mediation could be used to provide a layer of adaptation to allow run time substitution of services. A group of substitutable services would have to be nominated in advance using a human decision making process since it is unlikely that decisions involving names could be made by a machine, unless some form of a priori standardisation has been performed.
Chapter 2: Literature review

Protocol Level
Most distributed computing environments use a standard protocol for communication. The web services environment uses XML-based WSDL and SOAP protocols while Java RMI and DCOM use proprietary standards. In order to allow different systems to communicate with each other, differences in protocols would have to be resolved. GC-based mediation can be used to allow clients and services with different protocols to contact each other by providing adapters that convert the protocols. Incoming SOAP requests could be converted into tuples and conversely tuples could be converted into outgoing protocols suitable for web services, CORBA or Java RMI.

Conceptual Level
Conceptual heterogeneity refers to operations or data that have the same name but refer to different domain concepts [41], or the converse, where the names are different but the concept is the same. This form of heterogeneity usually must be resolved by human intervention. Once concepts have been resolved, mediation can be used to provide suitable adaptation at run-time.

2.7.3 Aggregation of Services

Most web services available today provide simple one-shot request/response style interactions. Therefore, the opportunity exists to create aggregated services by composing single purpose services into more complex value-added services that perform a variety of tasks under the one façade. One of the main challenges facing service aggregation is coordinating the interactions between the constituent services. This can either be achieved by using a centralised “engine” approach [42] or, in the case of a GC-based approach, the coordination information can be distributed among the components that make up the aggregated service.

The aggregation of services provides the opportunity to build complex composite services from basic services. In one scenario, hotel airline and car rental services could be composed into a holiday booking service. The services are modelled as tasks from a workflow plan and the aggregate service is created by managing the flow of control between the tasks. This concept is known as business process modelling and
there are tools available from many vendors for design and implementation of workflow systems (a brief survey is contained in [17]). Once a workflow design is implemented, unless the workflow execution environment can react to changes such as network latency or service unavailability the workflow may not behave as planned if problems occur. GC based mediation is a promising platform to implement workflow designs due to several factors:

- The decoupled nature of GC allows workflow tasks to be implemented as autonomous programs that communicate with other tasks using GC as the medium
- Tasks can be added or removed from the workflow dynamically and such changes are transparent to the other tasks
- Messages can be easily multicast to a subset of tasks or to all tasks by leveraging the decoupled addressing of GC

In the following sections, some examples of the application of GC-based computing are presented followed by an overview of the YAWL workflow management system [43].

**Workflow capability testing**

The normal process that would be followed in the development of a business process from a design into a workflow would be firstly to design the workflow and then to implement the workflow using a workflow management system. Once a workflow has been designed and implemented, any problems such as cyclic paths or deadlocks that may be inherent in the design cannot be determined until run time, perhaps with undesirable consequences. A workflow, which was implemented using GC-based techniques, could be designed to incorporate a “test mode” which could be used to investigate the potential pathways that the workflow could activate. This would involve sending a test “token” or message to the workflow, which in turn, would be replicated and sent to all of the down stream elements making up the workflow. Internal logic in the workflow elements could determine if there are any feedback paths.
Post Implementation maintenance

Top down design means that once a workflow has been designed and deployed, changes to the work flow can only be carried out by changing the design of the work flow, republishing it and then redeploying it [44]. However, because entities that interact in a GC-based environment are decoupled from each other and execute in their own thread of control, they can be removed and replaced without affecting other elements. With the correct timing this may even be accomplished while the workflow is running.

For example, in the simplest case where one of the workflow elements needs to have its functionality updated but there is to be no change to the workflow pattern, a simple replacement of the workflow element will suffice. The workflow may need to be stopped to do this. However, if the workflow pattern needs to be changed, then multiple existing elements may need to be updated and new elements added in order to achieve the desired effect. This is a good example of unanticipated software evolution (USE) [45] which has been the subject of some research. Workflow elements implemented in a distributed GC environment may be easier to maintain than elements implemented using a centralised approach because the distributed elements (cf. interaction machines) encapsulate their own logic and are physically decoupled from each other. For example, because the elements are separate entities, there is less chance of breaking any explicit or implicit dependencies.

Service Aggregation Environments

In recent years, many software tools have become available which allow workflow designs to be created using graphical user interfaces. An extensive review of most of the prominent Workflow Management Systems is covered in some detail in section 3 of the paper Workflow Patterns [17].

Service composition is very similar to other traditional areas of interest such as Business Process Modelling and Business Process Coordination. Service composition is basically the interconnection of autonomous heterogeneous services in a manner that gives a predictable outcome. There are many workflow tools on the market all trying to achieve this aim and some from high profile companies such as Lotus
(Domino workflow), Xerox (InConcert) and IBM (MQSeries) [17]. However one of the problems to arise from the plethora of systems is that there are very few standards and this leads to the problem of business processes being described differently by different workflow management system vendors. While workflows can be understood from the data perspective and resource perspective it is the control-flow perspective that gives the greatest insight into the effectiveness of a particular workflow [17].

Currently most workflow languages support the basic elements such as splits and joins but these are not implemented uniformly. One attempt to provide a thorough analysis of workflow systems from basic elements to complex has been carried out by Van der Aalst et. al [46]. Additional examples of a full range of workflow scenarios is available from the Workflow Patterns web site [46]. Our research will make use of these workflow patterns and also the workflow tool YAWL [47] which has been developed at the Centre for Information Technology Innovation (CITI) at QUT.

**The YAWL Editor Environment**

The YAWL editor [48] is a Java™ Swing based workflow design tool. The editor allows a user to create various elements and to connect the elements to create a workflow (see Figure 6). Once a design has been finalised it can be saved and exported to an XML format.

![Figure 6 Basic workflow design in the YAWL editor](image)
GC-based translation of the workflow

The XML description of the workflow must now be translated into a set of classes that fulfil the role of the tasks in the workflow. The translation process involves traversing the elements in the XML and extracting information such as tasks, conditions and flows. In a GC based environment, tasks are implemented as GC aware programs which execute autonomously and concurrently. Tasks communicate with each other using GC. The conditions and flow information which guides the workflow is programmed into the tasks during the translation. In order to move an aggregate service from the workflow design stage to the implementation stage the workflow has to be translated into the elements of the GC based environment. The workflow can be divided into three basic elements: tasks, conditions and flow, which are identifiable in the exported XML. The tasks are the operations that are carried by each element (e.g. book car hire), the conditions are applied to incoming and outgoing messages (e.g. if car is booked and hotel is booked then contact airline booking) and finally, flow is the intended direction of messages between tasks.

2.7.4 Other potential benefits of GC

While this research is primarily concerned with aspects of mediation and aggregation, there are several other aspects of service delivery that could benefit by the application of GC. Although these are outside the scope of this current research, they are discussed here for completeness. In addition, when taken together with mediation and aggregation aspects, this section goes part way in supporting the notion of GC as a basis for a holistic service environment.

“Pull” based processing
Clients can only address RPC-based services in a one-to-one manner and other clients, which require the same service, must wait until the service is free. This can result in delays and time-outs. In a GC-based environment, clients, messages and services exist concurrently in separate threads of control. Therefore, clients and services are decoupled from each other due to the buffering of messages in the space.
Because the message has an existence independent of the client and the service, messages can be picked up by any appropriate service that is active and monitoring the space. This introduces the possibility of one-to-many communications in the form of pull-based non-intentional multicasting.

This decoupling also helps shape the activities that may be performed by the GC-based participants. For example, 1) Clients send messages or tasks and are then free to carry out other processing while waiting for notification of a reply; 2) Messages or tasks are stored in the space until removed (a natural buffering) and 3) Services process messages or tasks and, when convenient, check for new messages or tasks to process. This leads to a number of benefits:

- Because the messages exist independently of senders and receivers, the messages may be processed by any service that can handle this type of message. This means that additional services can be added statically or dynamically to handle periods of high demand. Also, services can be replaced or upgraded without disruption.
The service can decide which messages to accept and when to accept them. As a consequence of this faster services may process more messages than slower services. This “pull” based processing can assist load balancing.

**Security**
While the ability to transmit executable code to a remote location sounds promising, security implications related to sending binary code to a remote location for execution would have to be investigated. Such remote execution of potentially malicious code could interfere with or damage the remote computer. For example, similar security aspects have been addressed to a degree with Java Applets and Java RMI by ensuring that executable code is partitioned to some extent from the computer by the Java “sand box” in the case of Applets and by the application of security policies in the case of Java RMI. This partitioning usually takes the form of a restricted instruction set, that is, instructions which read data from and perform operations on the host computer are restricted. When distributed behaviour techniques are applied to corporate intranets, security policies that are already in place and the trustworthy nature of these environments may be sufficient to allow unchecked operations to be performed. Security is a large topic and is outside the scope of this research.

**Capacity Management**
One definition of capacity management is “processes concerned with ensuring on-line response times” [49]. While originally relating to response times of network mainframe systems, capacity management is now being studied in relation to internet or web based services such as high demand web sites. Capacity management also has application for the reliable delivery of services.

Services that are under heavy use can become slow or unresponsive. A GC-based mediation system can be used to monitor the load that is being placed on services. In the case of remote services, excess requests can be redirected to alternate services if available. In the case of local services (services provided within the mediation framework), the mediation system can create and invoke duplicate services to handle high demand.
Chapter 2: Literature review

Active Mediation
Liu et al [50] proposes the concept of autonomous services that are linked together to compose software applications. Autonomous services have the following characteristics:

- Computationally intensive and asynchronous
- Run on distributed hosts
- Have heterogeneous interfaces
- Managed without central control

Liu proposes Active Mediation that allows clients of autonomous services to modify the behaviour of an autonomous service by sending dynamic routines to the autonomous service. Active mediation relies on the notion of mobile code [51] to achieve this. Benefits cited by Liu include the ability to move processing to a node where it will create the least network traffic. Liu’s concept of Active Mediation can be simulated in the AOS environment by using Active Objects as autonomous programs. Active Objects can be replaced dynamically at run-time or they can have their behaviour changed by sending them new or replacement routines as Java objects. This type of dynamic code “change-ability” could be important to a mediation system that needs to be maintained without downtime. Parts of the mediation system could be updated or replaced while the mediation system is still running.

Availability Management
Availability of services has a direct influence on reliability. While tolerable to a human user browsing web pages, unavailable services would hinder a machine-to-machine interaction since it may affect the operation of a system in unforeseen ways [52] perhaps causing the entire system to fail.

GC-based mediation can be used to create a register of available services. Availability can be determined either by polling remote services or by notification from native services. Unavailable services can be removed from the register and active services added. When a client calls for a particular service that is unavailable, the call is translated using heterogeneity resolution and redirected to a suitable available service.
Chapter 2: Literature review

Legacy Systems
One of the motivations behind web services was to allow organisations to expose their business functionality to a wider market place [53]. However, taking an existing system written in COBOL and modularising the business code could be a very difficult process [4]. Two approaches have been suggested [54]:

Converting legacy code to an OO language like Java
This may be an option for small to medium systems but the costs involved could be prohibitive for large systems. For example, if a system does not lend itself to easy modification then many changes may have to be made to incorporate a GC paradigm and a great deal of testing may have to be performed as a result.

Providing a mediation layer to convert incoming requests
This option allows for a layer of mediation that receives incoming requests and accesses selectively modularised business code. For this approach to be feasible, the business code would need to follow a modular architecture (ideally it would be object oriented). If the business code is not modular then it could be prohibitively expensive to take monolithic code and re-engineer it into modules.

From a different perspective GC based mediation can be used to provide backward compatibility to all existing services, not just services that expose legacy systems. Currently, services are available across the Internet in a variety of protocols, for example, web services, CORBA and CGI. GC based adaptation can be used to provide adaptation to these services.

2.8 Implementation Issues

The most efficient way to develop a GC-based service environment is to use one of the existing GC-based language extensions. While all GC-based languages support primitive operations similar to read, write and take, these alone are not sufficient for the creation of a service environment that supports advanced features such as mediation (a major focus of this research). One of the most important aspects of mediation is the ability to detect the presence, arrival and departure of messages in a space and while this can be done using the primitive operations, this would require
continuous polling of the space. Some languages such as JavaSpaces and Active Object Space (AOS) provide a notification mechanism which allows a “listener” to be notified when a particular message arrives and, in the case of AOS when it departs. Therefore, the selection of an appropriate GC-based language is important.

Background
A number of space-based languages have been developed over the years and have followed the example set by Gelernter and Carriero [37], that is, they are not languages in their own right but languages used within other languages such as C, C++ and Java. Gelernter and Carriero claim that a complete programming model can be built by combining a computational model (using traditional languages) and a coordination model (using space-based languages) and that the interests of these two models are orthogonal.

Languages such as space-based languages that coordinate activities have also been categorised by Arbab [18] as endogenous or exogenous. Endogenous languages rely on primitives which are within the computation module and exogenous languages such as Arbab’s Manifold provide primitives that effect coordination from outside the computation module. Arbab argues that endogenous languages lead to the mixing of computation and coordination semantics while exogenous languages separate coordination from cooperation and therefore lead to pure coordination modules that are easier to understand and reuse. However, our view is that the “mixing” that occurs with an endogenous approach is one benefit that leads to coordination entities that are self-contained and autonomous and can be easily updated or replaced without affecting other coordination entities. The following sections give an overview of some of the more prominent examples of space-based languages.

Linda
Linda, the first space-based language, was created in the early 1980s by Nicholas Carriero and David Gelernter as an alternative to other parallel programming approaches [30]. However, Linda is not a language in its own right but can be used within other languages such as C and C++ to provide GC capabilities.

42
JavaSpaces

JavaSpaces™, offered by Sun Microsystems [28] [16] is an implementation of the tuple spaces paradigm used within the Java programming language. Tuples or messages are written to the space as serialised Java objects. Reading or taking the message from the space requires de-serialisation. The objects written to the board can be passive objects, that is, they do not encompass their own thread of control [55] or they can be active objects. Passive objects can only be instantiated and their public fields inspected while active objects have methods that can be called to perform computation. An example of this is the generic worker pattern [16].

JavaSpaces provides the three basic operations or primitives (read, write and take see Figure 9) plus additional features such as notification and transactions.

![Figure 9 Overview of JavaSpaces™ operations](image)

All GC based languages rely on associative lookup for the addressing and finding of messages. JavaSpaces implementation of associative addressing relies on “templates” which are used to specify patterns of values to be matched in the messages. If more than one message matches the template then the message found will be non deterministic.
Active Object Space

Active Object Space (AOS) formerly known as JavaBoard, is a Java based implementation of the tuple-space or blackboard paradigm designed and implemented at the Centre for Information Technology Innovation at Queensland University of Technology [29]. Messages are written to the space as serialised Java objects and AOS supports the same primitive operations (read, write and take) as JavaSpaces but does not support transactions. Messages are found using template-based associative lookup similar to JavaSpaces.

AOS supports a notification mechanism similar to Java Spaces to detect the arrival of messages in the space but AOS adds notification for additional events such as messages that leave the space and messages that exist in the space when the listener first connects. The ability to detect the existence and movement of messages in a space is necessary to provide mediation facilities.

A feature of AOS not found in other implementations of space-based environments is Active Objects. While most messages passing through a space contain only data (passive tuples) Active Objects contain executable code. When an Active Object arrives in an AOS, the object can be automatically removed from the space by the AOS environment and executed in its own thread of control. This would allow a scenario where Active Object mediators or adapters could be sent to an AOS as required where they are executed on arrival.

T Spaces

T Spaces is IBM’s implementation of tuple-spaces [56]. T Spaces is implemented in Java and contains similar features to JavaSpaces and AOS but also has a number of unique features which are not of interest.

2.9 Summary

Much “popular” literature exists to support the case that web services has “taken the world by storm”. However, there is still enough ongoing research to suggest that web services is still in its early stages and the continuing evolution and emergence of web service standards would tend to support this view. While one of the main “selling”
points for web services is decoupling, there are some web service implications which stem from the use of RPC as a communication mechanism that have an adverse affect on decoupling at the low level. The limitations of RPC semantics are well documented and its intention, to provide remote calls with local semantics, could well be considered as orthogonal to the aims of web services.

GC is an alternative communication mechanism which has been designed, a priori, to promote the interaction of entities that are distributed across a network. GC provides two inherent forms of decoupling 1) spatial decoupling – because there is no reliance on network addresses for message delivery and 2) temporal decoupling – because all messages rendezvous in a buffer called the “space” before being removed by a recipient. A GC system developer must take into account issues such as asynchronous communication and decoupled addressing at the outset. A review of the literature however, reveals that while there have been a number of GC languages developed over the last twenty years the application of GC to the delivery of services is not common. One fundamental difference between RPC and GC is that RPC employs a “push” based messaging technique when sending messages while GC is intrinsically a “pull” based messaging system because messages are only received at the discretion of the receiver. Pull based messaging has two benefits; Firstly the service can decide when to receive a message and need not honour all requests; Secondly, and importantly for this research, “pull” based messaging implies that there is a buffer (or space) where messages rendezvous before being removed by a recipient and this introduces the opportunity to intercede in communications between senders and receivers. In this research, mediation is the mechanism to provide this intercession. RPC does not offer a natural means to intercept messages as they travel between endpoints.

Mediation offers a number of opportunities, not only to support existing service environments (backward compatibility), but also to promote extensions to existing service environments. In particular:

**Mediation for backward compatibility**

Chapter 3 focuses on protocol mediation which offers the ability to re-use existing services from various service environments by intercepting and adapting client
requests to match the protocol of the target service (eg. SOAP). Services register with mediation and the mediator creates an adapter that maps the service protocol (eg. SOAG or CGI) to a generic GC-based protocol that is used by clients.

**Interaction mediation**
Firstly, Chapter 4 focuses on extensions to the web service concept of UDDI registries by introducing a new concept called a Location Service. While a registry is part of the web service architecture, a Location Service performs the same function as a registry but is implemented the same as other services and is therefore not part of the architecture. Secondly, we examine an aspect of distributed behaviour allowing service providers to define an invocation interface for each service. The invocation interface can be a GUI object that is passed to the client so that the client can invoke the service without the need for interface level programming. The invocation interface could also be a stub generator or an API object.

**Aggregation**
Chapter 5 focuses on another extension to services: aggregation of simple services into more complex ensembles. Current web service interactions are mostly one-shot request/response style operations. While this is suitable for simple web service interactions, when the coordination of multiple web services is required over a period of time the underlying RPC semantics can result in limitations. For the delivery of services, GC based aggregated services offer a decentralised approach to service aggregation that is scalable and naturally asynchronous. Chapter 5 covers the design, translation and implementation of aggregated services.

**GC for a service environment**
As we have seen, the GC approach can provide many “pieces” to facilitate service interactions. In Chapter 6, we look at all interaction aspects supported by GC that can be pulled together to provide an encompassing environment for web services.
3 Mediation for backward compatibility

Mediation, in the context of this research, is the process of interceding in communications between clients and services that are interacting across a network. Mediation implies the existence of a third party (the mediator) which intercedes “transparently”.

There has been a number of prominent service environments introduced over the years such as for example CORBA, DCOM and CGI. More recently, web services has been introduced as a standards-based industry collaboration. However, all of these environments introduce their own system of protocols for interaction. While it may be sufficient for a new service environment to introduce a new way of enacting “services”, additional value can be attained if the new service environment also supports interaction with existing services which use a variety of protocols (such as those mentioned above). Hence, one use of mediation is to provide a degree of backward compatibility to existing services and this aspect is the focus of this chapter.

Backward compatibility

For a system to be backwardly compatible to a wide range of existing services, issues of heterogeneity must be addressed. That is, the system must be able to allow interaction between not only services with different protocols but also allow clients with different protocols to participate. In general terms, GC is just another protocol but in this research GC is the focus of attention. Therefore, there are three possible interaction scenarios which could be considered:

1. GC client interacts with SOAP/CGI/RMI services
2. SOAP/CGI/RMI clients interact with GC services
3. SOAP/CGI/RMI clients interact with SOAP/CGI/RMI services

Scenario 1 involves the creation of adapters that translate the calls from the GC client to the protocol of the service. Scenario 2 would be a reverse of the process in scenario
1. In the case of a SOAP client contacting a GC service an adapter would have to be created which was, in effect, a web service that translated calls into GC. Finally, scenario 3 is a combination of scenarios 1 and 2 and could be achieved by creating adapters on both the client side and service side of mediation (see Figure 1).

![Figure 10 Heterogeneous client/service interaction using adapters](image)

In this research we have chosen to explore scenario 1 and in particular to explore GC-client to SOAP-service interactions only. The reason for this is that having developed one service adapter, this could then be used as a framework for other service adapters using the same (or different) protocols. Also, the techniques used to create a service adapter could be re-used to provide client-side adapters.

Web services are not the only platform being used to deliver services across networks. Environments such as CORBA, CGI and even Java RMI have been in use for many years. However, by adding GC based mediation, services from these varied environments can be made available to a wider audience and accessed using a generic protocol. To achieve this kind of backward compatibility we require two concepts: A generic client that interacts with the GC environment using a GC-based protocol and an adaptor which converts the request from the client from a generic protocol into a request for the service in a protocol specific to the service (Figure 10).

The generic client uses a standard protocol defined by the GC-based environment. This has the benefit that service client developers do not have to become familiar with a number of protocols. Adaptors must be written to allow the standard protocol to be converted to each one of the service protocols. These adaptors then intercede between clients and services ensuring backward compatibility to services that use legacy
protocols. In this scenario, there are no changes to the way existing one-shot service invocations operate and the usual “conversational” aspects these interactions remain unchanged.

In Figure 11 another view is presented; requests for services from clients pass through the mediation layer where they are intercepted by protocol specific adapters. The adapters unmarshal the arguments from the generic protocol, convert the parameters and then re-marshal the arguments into the service protocol before passing the request on to the intended service. The response from the service is a reversal of this process. In effect, the GC client interacts with a remote service wrapped in a GC-based adapter.

For example, in order to allow a generic client to interact with a CGI service an adapter is created which converts the standard GC protocol to a CGI specific protocol. The main task of the adapter is to unmarshal the parameters received from the generic client, carry out any type conversions if required and then to marshal the arguments ready for the call to the CGI service. The response from the CGI service is handled in a similar manner. The adapter handles any CGI specific implementation details such as for example JAR files or DLLs.

All of the adapters follow a similar framework. However, a CGI adapter for example would use different JAR/DLL support infrastructure than say, a SOAP adapter and also would have different call semantics. Adapters that are being programmed for the same protocol would be basically the same but would call different methods and use perhaps a different number of parameters with different types.
In summary, the generic client is not dependant on the platform of the service (Java, .NET etc) but also is not dependant on the protocol of the service (SOAP, CGI etc). The adaptation allows for the re-use and “service-tisation” of previously existing disparate services in a manner that is transparent to the client, that is, the client is not aware that an adaptation has taken place. The generic client provides a consistent means by which developers can create clients that interact, not only with web services, but also with services from other platforms. The developer does not need to be concerned with the intricacies of SOAP or CGI etc. as the appropriate adapter has already catered for this. For example, the developer does not need to be concerned about the XML details of WSDL and SOAP and need only be concerned with the standard protocol adopted by the adapter. In other words, the developer only needs to learn one protocol (the standard GC-based protocol) rather than a suite of protocols.

The use of GC-based mediation introduces a number of concepts that benefit the service developer, the service user and the service environment. For example, current web service client implementations usually rely on web service frameworks such as for example, BEA WebLogic or IBM WebSphere for the development of not only web services but also web service clients. Web service clients developed within these environments may require many large support JAR files (or DLLs in the case of .NET) for the operation of the client. This introduces problems if the client programs are to be distributed since, not only would there be large files to distribute, but there may also be licensing implications.

On the other hand, GC based mediation promotes the creation of lightweight clients that need only a single small JAR file. The large support files are only needed by the adapters and can be located and managed centrally. Therefore, using a generic client and moving the implementation details to the adapter makes it easier to develop service clients. For example, depending on the API used, a SOAP adapter may require the use of many JAR files (eg. Apache tomcat comes with 24 JAR files). If we were to create a SOAP client, support files such as these would have to be distributed to each site where the client was to be used. Using a GC-SOAP adapter allows the files to be stored centrally with the adapter thereby centralising the administration of these files and reducing the complexity of the client installation.
3.1 **Mediator architecture**

In this section we present a mediation architecture that can be used to make existing services available to a generic GC-based client by using mediation. Mediator operation can be divided into three sections as depicted in Figure 12. Firstly, *Services* are a collection of heterogeneous, network accessible entities that use various existing protocols. Secondly, *Mediation* intercedes in communications between clients and services and uses Active Object Space (AOS) to facilitate this intercession. Finally, *Clients* represent any AOS enabled program.

**The register process:** It would be possible to access a UDDI registry to get information about suitable services, but suitability is assumed to be determined by human interaction. Once each service is registered, an adaptor is programmed into the mediator. In a web services environment a similar process must be undertaken; when a service requirement emerges, the developer of the service client must also search one or more UDDI registries to find a suitable service. If, on the other hand, a GC-based service was to be published in a UDDI registry then an adapter would have to be created inside a SOAP wrapper and then the SOAP interface published to a UDDI registry, the same as any other web service. This process is similar to the publish process in UDDI.

**The advertise process:** Once a service registration is received and the adapter created, the service is advertised by placing a service description object into the space. This object can be retrieved by a prospective client and used to invoke the service. The object contains information about the service methods. This process is not currently represented in the UDDI model.

**The find process:** Clients can see all advertisement objects lodged in the space. If a particular advertisement is of interest, it is read from the space and the information about the service such as service type and description are examined. If the client knows what type of service is required (eg. printers) the client can search by associative lookup to find only printer services. Search scope can be narrowed or widened as desired. This process is similar to the subscribe process in UDDI.
The invoke/response process: Once the appropriate service description is found, details about the service (methods, parameters and parameter types) are extracted from the service description object ready for invocation. The client invokes the service by passing messages to and receiving messages from the space. The adapter that was created by the mediator converts the messages to the service protocol. This process is similar to the bind process in UDDI.

In summary, the generic client invokes services through mediation. No direct interaction is required with the service. This promotes a service environment that not only supports the creation of new GC-based services but also allows existing investment in services in other environments to be leveraged. The disadvantage is that an adapter must be written for each service and a number of protocols may have to be mastered.
Client/Service interaction
In Figure 13 the interaction between clients, mediation and services is presented in terms of GC-based messages that pass between these entities. In the AOS environment GC-based messages are implemented as Java classes and are passed as serialised objects:

**Service Description Object (SDO):** The SDO encapsulates information about the service such as method name and parameters. In this part of the research the SDOs are manually created for a small set of candidate services. However, in Chapter 4 the SDOs are created by the service provider and sent to a location service to enable the service to be discovered by clients.

**Service Request Object (SRQ):** The SRQ is provided for the service at the same time as the SDO. The SRQ is used to encapsulate the data passed from the client to the service during invocation.

**Service Response Object (SRP):** The SRP is created by the service after the service has been invoked. The SRP encapsulates the response data to be returned to the client.
The three key architectural elements are now discussed in detail:

Each service that is to be used with the mediator must have an adaptor programmed for it. Each adaptor provides a bridge between the GC-based AOS environment and the environment in which the service exists. For example, in order to provide an adaptor for a web service, the mediator developer would create an AOS Java based adaptor that uses the Java SOAP APIs such as JAAS to contact the web service. The mediator makes services accessible to clients via the AOS by deploying Service Description Objects (SDO). Clients access the services by taking an SDO from the mediator and using information in the SDO to interact with a service via the mediator. The client does not contact the service directly but uses a channel provided by the mediator (although direct contact could be implemented if desired). Typically, clients will retrieve and examine a SDO and if suitable will then retrieve SRQ that matches the SDO. The service parameters are then entered into the SRQ and it is sent to the mediator for processing.

The mediator uses the AOS “notify” mechanism to wait for SRQ objects to arrive. Once a SRO arrives, the mediator takes the request from the space and, depending upon the type of request, contacts the appropriate web service. Processing the request is where the adaptation is done. This is where the adaptor un-marshal the arguments from the request and adapts them before contacting the web service. Once a response has been received from the web service the result is packaged in a response object ready for collection by the client.

The client can be any AOS enabled program operating anywhere on the Internet, however, an existing client could be used providing it was Java based and had access to the AOS JAR file. Once connected to the mediator, the client uses the read operation to acquire a SDO and uses the information contained therein to get a matching SQO. The client then uses the SRQ to invoke the service. If the client was required to carry out some other operations while it was waiting for a response then it could use a call-back provided by the AOS notify mechanism.
Thus, mediation allows for protocol decoupling between clients and services since clients will always use GC-based protocols irrespective of which type of service is being accessed (eg. WS, CGI, RMI or GC-based).

### 3.2 Proof of concept prototype

During the course of the research, prototype components were created in order to test the feasibility of providing mediation based backward compatibility to existing services. There are two GUI components to test aspects of protocol mediation. The mediator window performs the activities of the mediator (Figure 13). The client window is used for testing client operations but could be replaced by any AOS aware program. In this case, we have AOS specific clients accessing web services via SOAP messages.

#### 3.2.1 The mediator window

In this section, we present a prototype of a mediator used to test the architecture described in Figure 13. The main purpose the mediator is to access a list of services from a registry (in this case XMethods) and then allow the operator to publish any of these services in the form of a service description object (SDO). This SDO can then be accessed by a client and converted to a service request object (SRQ). The client fills out the details (parameters) of the SRQ and sends this to the mediator which adapts the SRQ to the SOAP environment and calls the service. When a response is received the mediator converts this into a service response object (SRO) and returns this for collection by the client. The list of services displayed in the list box at top right is obtained from XMethods [57] by calling a service that returns a list of all services. XMethods provides a large variety of web services that can be used for experimental purposes. Each service has its own WSDL signature and this is retrieved from the service and displayed in the list box at the lower right when each service is selected.
Chapter 3: Mediation for backward compatibility

Figure 14 Services Mediator window

The scenario depicted in Figure 14, has created an adapter for the California Traffic Conditions service and the mediator has published a SDO for this service. Messages that occurred during operation are listed on the left.

Figure 15 Web Services mediator interaction

Figure 15 summarises the mediation interaction. In the mediator window an operator loads a set of service descriptions (1). When a service is selected an adapter is created
and the service is published to the mediator (2,3). The client then discovers the service description and invokes the services (4). The client interacts with the service via the adapter (5,6).

### 3.2.2 Generic client window

The generic client window is used to interact with the services that are made accessible by the mediator window and can reconfigure its visual interface depending on the service that is selected. This removes the need to program changes to interact with different services. The example shown in Figure 16 has selected the California traffic service and shows traffic information for highway 80 (this was made available by the mediator – see previous section).

![Figure 16: An example of a generic client window](image)

Any SDOs that have been made available are listed at the top left. When a service is selected, the descriptions of the methods, parameters and their types are extracted from the SDO and displayed for user entry at bottom left. In the case shown, we are requesting California traffic conditions for highway 80. The information is then converted into a SRQ and sent to the mediator for adaptation to the remote service protocol (SOAP). When the SRO is available the result of the web service call is displayed (right).
3.3 Analysis of backward compatibility and mediation overheads

Currently, most service interactions occur using a single protocol; for example, SOAP-SOAP or CGI-CGI. This means that services written in one protocol are not naturally available to clients that use another protocol even though they share the same network. In a GC-based environment, decoupling at the protocol level can be introduced to further decouple clients and services from each other (i.e. beyond the implementation decoupling introduced by SOAP/WSDL for example). This “protocol decoupling” can be achieved by providing a protocol specific adapter for the client to access and also by providing a protocol specific adapter to call the service. GC provides the common connectivity between the adapters. For example, Figure 17 shows an interaction between a SOAP client and a CGI service. The client adapter converts the incoming call from its native protocol into a GC message which is addressed to the appropriate service adapter. On receipt if the message, the service adapter converts the GC message into the native protocol of the target service. This process is reversed when the response is returned.

![Diagram of heterogeneous interaction using adapters](image)

The ability to provide for the invocation of services between heterogeneous clients and services has a number of benefits. 1) Existing services can be re-used by a wider client base. 2) Newer clients can access “older” existing services leading to backward compatibility. 3) The re-use of existing services leads to reduced service coding effort. On the other hand, the disadvantage is that a client adapter and a service adapter must be created for each protocol and service that is to be handled. This would require a degree of coding effort. Also, the programmer would have to be familiar with a variety of service protocols.

In the following sections we analyse some of the issues involved with creating adapters that provide heterogeneous client-service interaction and give some typical
code examples. In doing so, we only consider a SOAP-GC-SOAP interaction. Other protocols such as for example CGI and RMI would be handled in a similar manner.

**Client access adapter**

The client requires the ability to locate and invoke services. If the client is a GC based client then this can be done by interacting directly with the service environment using the appropriate GC primitives to read, write or take messages. However, if the client is not GC based then a client-side adaptor must be coded to convert the calls from the client protocol into GC based calls. The client side adapter operates within the service environment and appears to the client as a service that uses the same protocol as the client. For example, to allow a SOAP client to interact with the service environment would require an adaptor to be created that took incoming SOAP requests and translated them into GC primitives. In effect, a SOAP web service would need to be created to allow SOAP access to the services offered by the service environment.

Figure 18 shows an example of a SOAP client interacting with a SOAP service via the service environment.

![Diagram of SOAP to SOAP interaction](image)

In this example, a SOAP-GC adapter is provided to connect to a location service. Other SOAP-GC adapters would be provided as additional services are added. A SOAP-GC adapter is essentially a web service that translates client calls to the service
into GC-based messages. A SOAP-GC adapter could be built using one of the many freely available web service development kits. For example, using the Apache Axis platform a simple program can be written (using the JWS tools) that redirects the incoming SOAP request to GC-based location service and then returns the result to the SOAP client (See Figure 19).

```java
// ServiceLocator.jws – runs under apache axis as a web service
public class ServiceLocator {
    public String findService(String serviceType) {
        // open a connection to the space
        JavaBoard jb = new JavaBoard("123.456.789", port);
        // create a template to find a location service
        Template template = new Template();
        template.setClassName("GCLocationService");
        // find the service, load it and execute the query
        AugmentedObject ao = jb.read(template, 1);
        GCLocationService gcLS = (GCLocationService) ao.loadObject();
        String serviceName = gcLS.query(serviceType);
        // return the name to the web service client
        return serviceName;
    }
}
```

Figure 19 Example code for a SOAP-GC location service

The service in Figure 19 can be invoked by a SOAP client using the following URL:

Service access coding
As shown in Figure 18 a service side adapter would also be required for a SOAP-to-SOAP interaction. The purpose of this adapter would be to translate GC-based calls into calls to SOAP-based web services.

```java
// get the GC request that the client has sent
WebServiceRequest wsRequest = (WebServiceRequest)ao.loadObject();
String[] params = new String[wsRequest.params.length];
for (int i=0; i<wsRequest.params.length; i++) {
    // get the params from the request
    params[i] = wsRequest.params[i];
}

// interact with web service here
WebServiceResponse wsResponse = null;
Object response = null;
try {
    // bind to the service
    IProxy proxy = Registry.bind(wsRequest.wsdIURL);
    // make the call
    response = proxy.invoke(wsRequest.method, params);
    // parse the SOAP response into a GC based response
    wsResponse = new WebServiceResponse(
        wsRequest.service,
        wsRequest.id,
        response);
} catch (Throwable t) {
    // exception thrown – create an error response
    wsResponse = new WebServiceResponse(
        wsRequest.service,
        wsRequest.id,
        t.toString());
```
3.4 Summary

In this chapter we presented our concept of protocol mediation where the decoupling of clients and services (a benefit already established in the web services paradigm) is extended by allowing a generic client to interact with SOAP and WSDL based services. This is achieved by providing mediation-based adaptation that intercedes between clients and the services. The main benefit of this adaptation is the ability to reuse existing services that use a variety of protocols. Other benefits include:

1) Mediation-based adaptation handles the intricacies of the service protocols (in the example presented SOAP and WSDL) which means that developers of clients for mediation based services need only learn the one protocol (the protocol of the mediation based services).

2) Traditional web service clients can require a supporting infrastructure depending on the protocol supported by the development environment (eg. BEA WebLogic, IBM WebSphere etc). Mediation based adaptation moves the reliance on the protocol specific environment to the adapter thus reducing the reliance of the client program on support infrastructure. There are also benefits in licensing and supporting the infrastructure because it can be shared with any service that uses the same protocol.

In the next chapter, we look at further backward compatibility by extending the notion of the web services UDDI registry and we also look at the passing of behaviour as well as data between clients and services.
Chapter 4

4 Interaction Mediation

In the previous chapter, we focussed on a data-centric view of interactions between clients and services, that is, the mediation and adaptation of protocols during interactions between clients and heterogeneous services. In this chapter we expand our focus to include a behaviour-centric view of interactions by introducing the notion of a GC-based location service (similar to a UDDI registry) as an extension of GC-based services. We also investigate the concept of GC-enabled behaviour passing and how this can be used to further decouple clients from services at the interface level. The chapter concludes with an example of a simple aggregation of services to demonstrate these aspects and also to serve as an introduction to the next chapter.

4.1 Introduction

Web service interactions are based on the publish-subscribe-bind (PSB) paradigm. Service developers create services and publish them to well known UDDI registries. Service users find these services by manually browsing registries (subscribe) and then interact with the services directly (bind) using information from the subscribe process. PSB is a part of the architecture of the web services environment.

In this chapter we examine an alternative means of publishing service descriptions and subscribing to services by introducing location services (an extension of GC-based services introduced in Chapter 3). Location services are similar to UDDI registries in that their primary purpose is to allow service developers to publish services and to allow service clients to find services. However, the application of GC techniques to location services introduces the opportunity for two departures from the UDDI/PSB concept. Firstly, a location service is not part of the GC architecture and it can be created, deployed and used in the same way as any other GC-based service. The benefit of this is that the opportunity to create location services is opened up to all participants in the service environment (rather than just a small number of commercial registries eg. IBM, Microsoft, SAP, XMethods etc). Secondly, because GC is a pull-
based communication medium, location services can advertise their presence to prospective services and clients alike – an aspect that is not part of the PSB paradigm. While both of these aspects lead to a more open and malleable environment they also do not preclude the re-use of currently available UDDI registries, thus also introducing a degree of backward compatibility and expanding the current notion of the UDDI registry.

Location services can also take advantage of Java-based GC by allowing behaviour in the form of interface objects to be passed between services and clients. An interface object is a serialised Java object that primarily contains information about the interface of the service from which it originated but importantly can also contain behaviour in the form of methods that can be invoked by the recipient. A typical use of a method would be to send a request to the service from which the object was received.

The interface object can take three forms: 1) a graphical user interface which can be used for human interaction directly with the service, 2) a program which the client programmer uses to generates stub code with which to call the service and 3) an API object which a Java program can use to interact with the service. The ability to pass interface objects can reduce the need to hard-code interfaces between clients and services, which goes some way to resolving conceptual heterogeneity. For example, if two similar services have different parameters then passing a GUI interface object to the client allows the client to invoke the service without knowing the technical details of the interface. In order to facilitate the notion of location services and behaviour passing we deal with three concepts in the sections that follow:

- **GC-based services**: Services that can interact directly with a GC-based environment to 1) facilitate the mediation process 2) provide adaptation to heterogeneous services and 3) facilitate a homogeneous environment that has a malleable/open architecture
- **Location services**: A location service 1) is an implementation of GC-based services that extends the concept of the UDDI registry to provide registry-like services that are flexible and dynamic and 2) Expands the concept of naming and discovery to allow “plugging-in” of several mechanisms
• **Distributed behaviour:** Leveraging Java based GC environments such as AOS by passing executable code between participants. One potential benefit of this is the opportunity to eliminate the development of client side service interfaces which helps aid end user understanding and goes some way to aiding the resolution of conceptual heterogeneity in this particular area.

### 4.2 GC-based Services

Web services are usually associated with the concept of functionality which is available across the Internet. However, to simplify the discussion we will look at intranet examples including printer and fax management, and office tools such as spell checking which can be implemented as local GC-based services. A GC-based Service has several key properties:

- A GC-based service is closely associated with an AOS and interactions to and from the service occur through the AOS. There is no need in this case for protocol adaptation.
- A GC-based service can interact with other distributed computing environments such as for example Web Services, CGI or Java RMI see Figure 21

![Figure 21 Overview of GC-based services](image-url)
Chapter 4: Interaction mediation

As shown in Figure 21 GC-based services can be used as proxies or adaptors to connect to network based devices such as printers and faxes. Other GC-based services such as location services (discussed next) can provide a service in isolation (ie. without accessing other services or devices).

4.3 Location services and distributed behaviour

The web service environment is currently based on the Publish, Subscribe and Bind (PSB) paradigm which is a specification of how the independent entities clients, services and registries must interact (Figure 22). Given this standards based approach, the existence and operation of the registry is mandatory and the registry assumes a place as part of the web services architecture.

![Figure 22 The publish, subscribe and bind paradigm](image)

One example of the chain of events that leads to a PSB style client/service interaction is as follows:

- Clients, registries and services come into existence independently
- Addresses of registries are published by email or web pages to become well known
- As services are created by developers they are registered with a known registry (Publish)
- Designers of web service clients search registries for appropriate services and create clients based on the interface specifications of these services (Subscribe)
• Web service clients connect to services (Bind) that have been discovered in the service registries

During this process, all aspects of interaction are carried out using static addresses that must be known before interaction can occur. The PSB paradigm has two associated aspects. Firstly, a new registry wishing to enter into this process must become known, and secondly, registries do not interact with other registries.

In a GC based service environment, there is no need for an inherent concept of a registry and thus a registry is not required as part of the architecture. In a GC based environment, existing UDDI registries could be supported using protocol mediation as discussed in Chapter 3. However, since the ability to find services is important and quite specific, a new concept we call a location service is introduced. Some of the important features of locations services are:

• Location services are implemented and accessed the same as any other GC-base service, therefore, they have no special requirements.
• The method a location service uses to perform registry-like functions is not mandated. One location service may use SQL like queries and another may use simple string matching.
• Location services are free to utilise other services (even non GC-based services) for the purpose of resolving the query or in fact to delegate the query to another location service.
• Location services may utilise a similar discovery mechanism to UDDI and WSDL or actively advertise their presence to GC-aware clients and services.

In summary, location services have the same primary function as UDDI registries, that is, the discovery of services. However, due to their GC-based underpinnings location services are able to provide not only backward compatibility to existing UDDI registries but also features which go beyond the publish, subscribe and bind paradigm. Specifically, they may deliver behaviour to client sites which facilitates usage of the service – this may be targeted at specific user types (eg. novice vs experienced) and can be specific to a type of definition/query language.
4.4 Client Implementations

Any AOS aware client can interact with an AOS implementation. In addition, any WS aware client can interact with an AOS implementation by using an AOS aware Web Service mediator.

![Diagram of AOS based client/service interactions](image)

Figure 23 Comparison of AOS based client/service interactions

Participants in an AOS based interaction can not only pass data to each other but can also pass behaviour in the form of methods in Java objects (as discussed in the section on distributed behaviour). This capability can be used to further decouple participants from each other by allowing services to pass their interfaces to clients in the form of a Graphical User Interface (GUI) component. The client need know nothing of the interface for the service since the entire interface is encapsulated in the GUI object retrieved from the service. For example, in order for a client to interact with a remote service, would require the user to create a GUI on the client side which displays the interface parameters and allows the client to enter a set of values before invoking the service. In Web Services, the parameters can be derived from the WSDL and this is usually done by a tool such as WSDL4JAVA which generates a class (or stub) based on the WSDL.
In the AOS case, however, the location service allows the service to also publish a GUI object which contains the interface definition. This interface object can then be retrieved by a client, invoked and the parameters entered by the user. The object is then returned to the service with the encapsulated parameter values. One benefit of this approach is that the interface can be changed by the service provider without requiring that clients reconfigure their own locally built interfaces. A generic approach to this method is shown in Figure 25.

This approach was implemented in the research:

1. Each different type of location service would normally have a unique interface with which a potential client could interact. An AOS based location service can be used to pass an object which represents the interface of the location service.
2. Each different type of service would have its own invocation interface

### 4.5 Examples of use

Say a service provider wishes to provide printing services. The service provider creates a service invocation program (SIP) that contains a GUI that can be used to
invoke the printing service. The SIP allows the user to enter details such as duplex, colour and number of copies and also allows the user to attach a document for printing. This SIP is self-contained in that it encapsulates all of the information that is required to interact with the printer (duplex etc) and, importantly, all of the information required to send a print request back to the printing service from which it originated. The user of the SIP will not have to deal with details such as network addresses and interface parameters (protocol issues).

When the SIP development is complete, the service provider will find a suitable location service with which to publish the service. When the service provider publishes the SIP he/she will not only publish a description of the service but also the SIP as a serialised Java object. This means that any potential user of the service will have access to the SIP from the location service.

As mentioned in the previous section there are additional possibilities for client-side interaction with a service. Firstly, if the SIP contains a stub generator the client can invoke this to create Java stub code which matches the interface of the service. This can then be incorporated into the client code in a similar manner to Java RMI. This approach would allow machine-to-machine interaction to occur. Secondly, if the SIP is a de-serialised Java object then the client can call the methods in the object using reflection to interact with the service. One problem with this approach is that the object is only transient and could not be used outside of the thread of control of the program that is using it.
4.6 Overview of AOS based interaction

The diagram in Figure 26 shows how service environment elements (clients, services and location services) can be combined together into a typical scenario. Location services (LS1 and LS2) provide service publishing facilities for services and service location facilities for clients (cf UDDI registry). Typically, printer services (Service1) would be located locally while remote services such as existing web services would be accessed via a web service SOAP adapter (service 2). Providing a custom service or business process adapter (service 3) can include business processes from legacy systems.

![Diagram of AOS based interaction](image-url)
4.7 Implementation of AOS based interaction

The diagram in Figure 27 (next page) shows a typical interaction between a service, a location service and a client. Numbers shown thus [n] refer to events that are shown as arrows on the diagram. The scenario of publishing, finding and using a service can be broken down into the following phases:

Advertise: The location service writes a location service description object (LSDO) to the AOS [1]. This object can be read by any interested party and remains available for other service providers to read.

Publish: The service provider reads the LSDO [2], fills out the description of the service and returns the LSDO to the AOS [3]. The location service takes the LSDO [4] when it arrives in the AOS. The location service also writes a location service query object (LSQO) to the AOS [5] ready for client queries.

Subscribe: The client reads a LSQO [6] and fills out the details of the service that is required. The client then writes the LSQO back to the AOS [7] where it is picked up by the location service [8]. The LSQO is used by the location service to perform a query based on the client’s criteria. When a match is found the location services add the information about the service to the LSQO and returns it to the AOS [9] where the LSQO is picked up by the client [10].

Bind: The client uses the information in the LSQO to create a job description object (JDO) for the specific service and this is written to the AOS [11]. The service picks up the JDO [12] executes the requested service and returns the JDO (containing any result) to the AOS [13]. The client picks up the completed JDO [14].
Figure 27 Interaction between clients, location services and services
4.8 Simple aggregation of services

The ability to aggregate simple services into more complex services is a desirable aspect of any service oriented architecture and this will be explored more fully in Chapter 5. However, during the course of this phase of the research it was decided to explore the creation of simple sequential aggregated services using GC-based services. Figure 28 shows the concept of creating a simple aggregated service by passing a document first to a spell checker service, then to a fax service and finally to a printer service.

![Simple sequential aggregated service](image)

In order to test this simple sequential service aggregation a prototype management console was created Figure 29. The purpose of this console is to firstly to allow repeatable creation of the aggregated services that are used for the test (in this case this is the printer, fax and spell checker services) and secondly to allow the operator to monitor the progress of the interaction (i.e. to provide some form of visual feedback of the progress of the test). The services that make up the aggregated service are programmed to take incoming messages, process the message and then forward the message to the next element of the aggregation (In the case of this scenario this is as depicted in Figure 28. When the services are started they place a service description object in the space. When the client is started it finds the service description objects and lists the services that are available.

In the scenario depicted in Figure 28, a document is to be carried with the message for the purpose of being processed by each of the services in turn. When the message reaches the last service a response is returned to the client.
Figure 29 Service Aggregation Prototyping console

The web service architecture is based on the publish, subscribe and bind (PSB) paradigm. The goal of the section of research was to demonstrate an alternative to PSB that leverages the decoupled addressing that is provided by Generative Communications (GC). PSB binds service producers, service registries and service consumers to fixed network addresses but GC uses a system of name based associative addressing to allow 1) service producers to publish services to one or more potential consumers and 2) service consumers to find services of interest and to invoke them.

In the web services environment UDDI registries are part of the architecture but in a GC based system registries can be created as services which are therefore not part of the architecture but are more a specialised artefact.
4.9 **Summary**

In this chapter we looked at a further extension to the web services paradigm by introducing the concept of location services. Location services are similar to UDDI registries but unlike their web service counterparts they are not part of the architecture but are implemented in the same way as other services. We also investigated the creation of GC-based services to perform functions such as printing and fax services and how these can be integrated with location services into an integrated environment. Behaviour movement (passing of executable code such as a GUI or stub generator) was also investigated in order to help to resolve (to some extent) issues related to conceptual heterogeneity. We concluded by looking at a case of simple service aggregation and presented a proof of concept prototype as a precursor to a deeper investigation of aggregation in Chapter 5 and also the proposal for an integrated service environment in Chapter 6.
5 Aggregation

In the previous two chapters we focussed on two aspects of mediation 1) the mediation-based adaptation of various protocols to provide a degree of backward compatibility to existing services and 2) the promotion of mediation-based extensions to the web services paradigm; an example was the location service, an extension to the UDDI registry that also supported existing UDDI registries. We also investigated services from a distributed objects point of view by examining the benefits that can be achieved by passing behaviour between clients and services. While mediation is mostly concerned with the interaction between individual participants in a service environment, aggregation is more concerned with the collective coordination of a specific set of services to create a single more complex service. Service aggregation is a large topic and in this chapter we look at building on the concept of web service environment extensions (such as mediation and adaptation) covered in the chapters 3 and 4. In particular, we consider how GC based aggregated services can be designed, transformed into executable code and implemented in the AOS environment. To simplify our task we equate aggregation to that achieved via a workflow and utilise the YAWL language [43] to this end.

5.1 Overview

Currently, most web service clients interact with simplistic one-shot, request/response style services [24]. In order for a service client to gain benefit from the aggregation of multiple services the client would have to programmatically combine and coordinate the interactions between the services. Aggregation, in the context of this investigation, is this process of combining and coordinating multiple services into a single ensemble. However, in our investigation, this is not carried out at the client, but the aggregation is performed by the GC-based mediation environment and is therefore potentially available to other clients wishing to call the aggregated service.
Aggregated services are created here by using a workflow designer to visualise the flow of control between the services that make up the aggregated service. There are many advanced workflow design tools available that provide graphical user interfaces and drag-and-drop functionality to designers and a limited survey of these tools was carried out in [17]. Most workflow editors allow the designer to create workflow tasks, assign properties and conditional logic to the tasks and to connect the task to define the flow of control. For the purposes of this research we have chosen to use the YAWL editor [48]. The YAWL editor allows the designer to create a workflow plan and to set up the interactions between workflow tasks.

Workflow-based aggregation allows services to be modelled as a set of tasks that interact with each other to carry out the intention of the aggregated service. In a GC-based environment the tasks are implemented as GC-based services (AOS active objects) that contain coordination logic from the workflow or are used to contact remote services (usually both). GC provides some aspects which can be leveraged for the provision of aggregated services. Firstly, name-based addressing allows tasks to communicate with one or more other tasks in the aggregated service. The decoupled nature of name-based addressing allows communication channels between tasks to be easily changed at run time and also allows messages to be multicast to all of the tasks in an aggregated service if required (see cancellation later). Secondly, GC-based tasks are naturally asynchronous, which makes them a good fit for the concurrent communication required by tasks in a workflow based aggregated service.

Once the YAWL design is complete, the specification is exported to an XML format where it is translated into a set of AOS Active Objects that perform the activities of the tasks in the workflow. The Active Object tasks coordinate interactions with each other autonomously and communicate using GC. This process is the focus of this part of the research and is explained in more detail in the following sections. Traditional workflow management systems usually rely on centralised management where coordination of the workflow elements is carried out by supervisory software (similar to an operating system) sometimes called an engine (eg. the YAWL engine). This type of coordination model has been referred to as exogenous [18] since the operations that affect the coordination of the workflow elements are performed from outside the workflow elements. However, in a GC based workflow implementation the
coordination responsibilities can be distributed among the workflow elements. This creates a coordination model that is endogenous [18] since all of the operations which involve coordination are to be found within the elements implementing the workflow themselves. This leads to a distributed rather than a centralised control model.

In summary, the process starts with a design of an aggregated service as a workflow plan using the YAWL editor. When complete, this plan is exported into an XML format ready for translation into a set of Java classes (tasks) that implement the intention of the aggregated service. The Java classes communicate with each other using GC as the communication medium. When the classes are invoked messages are passed between the tasks based on the design of the workflow. All of the logic that manages the aggregated service is embedded in the tasks (ie. distributed among the elements that make up the aggregated services). Therefore, the combination of GC and GC-based classes provides a pseudo workflow engine.

Let us now look at a simple example (Figure 30) to show how single purpose services can be aggregated to create a more complex service. Consider for example, the aggregation of flight, hotel and car hire services into a holiday booking service. When the aggregated service is initiated, it concurrently contacts the flight, hotel and car hire services. The results from the three booking services are then combined and returned as a single response.

Figure 30 Example of simple service aggregation

This type of scenario can be implemented using AOS by creating GC-based services that encapsulate the activities of the tasks. The tasks then interact with each other...
using GC. Since the GC-based services are autonomous and decoupled from each other, interactions can be flexible and dynamic. That is, since there is no static address based connection between tasks, tasks could be replaced and/or updated without disrupting the workflow.

These aspects highlight the ability of GC to provide a natural communication medium for concurrently executing entities such as tasks in a workflow based aggregation. The buffered nature of GC is particularly useful for the asynchronous communications required for concurrently interacting processes while the associative addressing aspect of GC provides a flexible means directing communications between interacting parties.

5.2 Process Overview

When the design of an aggregated service (the workflow plan) is complete the design is exported from the YAWL editor to XML. However, the XML must then be analysed before being converted into a form suitable for the coordination environment. In the YAWL system this conversion is done by the YAWL Engine [42]. From the perspective of this research the XML must be translated into AOS active object classes and there are two considerations. Firstly, the workflow flows from the upstream initiation point to the downstream termination point. The information about the downstream flow can be easily determined by traversing all of the branches and nodes in the directed acyclic graph (DAG) specified by the XML. Secondly, the downstream elements need to be aware of the presence of upstream elements upon which they depend for decision-making information. This will require a data structure to track dependencies. Both types of information will be needed to generate coordination elements that can effectively execute the purpose of the original workflow.

Workflow systems in general are not without their limitations. One seemingly serious problem that has been dealt with in a number of ways by various vendors is the OR-Join problem [58]. An OR-Join is a conditional construct that must wait until a response arrives at all of its inputs before making a decision and creating an output result. The problem with the OR-Join is that there may be times where a decision
cannot be made because an upstream process on which a decision depends has not been activated. The diagram in Figure 31 illustrates a typical OR-Join scenario.

![Figure 31 Example scenario for the OR-Join problem](image)

If OrJoin2 is waiting for events from OrSplit2, OrSplit3 and OrJoin1, a wait-state will arise if ORJoin1 is not activated by either OrSplit2 or OrSplit3. The OR-Join problem and other workflow issues are discussed later in the appendix on workflow patterns.

### 5.2.1 Creating an aggregation - overview

Creating an aggregation in a GC environment is a three-stage process:

**Design:** The YAWL workflow editor is used to design the aggregation and to define the flow of control between the tasks.

**Translation:** The XML output of the editor is then parsed and converted into a set of GC-based service classes (AOS active objects) representing the tasks. Information about the flow of control and any external services to contact is embedded in each task. An aggregation manager is also created to initiate and terminate the aggregated service.

**Mediation:** If services that were specified in the workflow design are not available then mediation (discussed in chapters 3 and 4) is used to reconcile interface differences between the service that was requested and the service that was available.
**Potential service selection:** During execution of an aggregated service, tasks in the aggregated service may call other services to support the intention of the aggregated service. If these external services are not available then alternative services must be selected. This could be achieved by interaction with an appropriate location service.

**Execution:** The GC-based tasks are then executed in conjunction with the AOS environment.

#### 5.2.2 Designing aggregated services using YAWL

An aggregation design is created using the YAWL editor by adding tasks and connecting the tasks together to create a YAWL “Net”. Tasks can then be “decorated” by applying input joins or output splits. There are two forms of tasks available in YAWL, atomic tasks and complex tasks (which can contain an entire Net). For simplicity only atomic tasks are considered in this research. However, a GC-based approach to complex tasks would be possible by providing tasks that belong to a particular Net with an ID (ensuring that coordination only occurred between tasks with the same ID).
Chapter 5: Aggregation

Figure 32 Holiday booking service designed in YAWL

The screen shot in Figure 32 shows a YAWL net that has been set up for simple holiday booking service. In this aggregated service there are five active elements: the input condition, three atomic tasks (hotel, flight and car booking) and an output condition. The input condition initiates the service by sending a message to each of the booking tasks. When each of the tasks receives a message it carries out some processing based on the content of the message (this may involve the calling of a remote service). When each task completes, it sends a message to the output condition which merges or synchronizes the messages and perhaps carries out more processing before terminating.

5.2.3 Translation of the workflow

Once design of the workflow is complete, the YAWL editor exports a representation of the workflow design into XML format. The XML is structured in such a way that the elements and their relationship to one another can be extracted programmatically. Part of this research was the creation of a translator that can interpret the XML and create a set of GC-based classes that represent the tasks in the workflow. Each of these classes contains three concepts 1) the input algorithm which determines how the task should respond to the incoming messages, 2) the internal process (the task to be carried out) and, 3) the output algorithm which determines which messages are to be sent when the task is complete. All of the information for these three areas is determined from the XML and is programmed into each task as it is translated. The
XML is read using a SAX XML parser which, during processing, looks for special nodes embedded in the XML. The processControlElements node contains a set of child nodes (task nodes) that define the tasks in the workflow (Figure 34). The task nodes contain all of the information about each task including which tasks it flows into and which tasks it waits for. The following pseudo-code (Figure 33) shows the steps followed to gather the task information and create the GC-based task classes.

```java
Task[] taskList = new Task[]
For each (taskNode) {
    // create a new task for this node
    Task currentTask = new Task(taskNode)
    // add to the task the input and output conditions
    // that have been defined in the taskNode
    currentTask.addInputCondition(taskNode)
    currentTask.addOutputCondition(taskNode)
    // for each task that this node flows into
    // add this to the current task
    for each (taskNode.flowsIntoTask) {
        currentTask.addFlowsInto(taskNode.flowsIntoTask)
        // if this task is not already in taskList add it
        if (not taskList.contains(taskNode.flowsIntoTask))
            taskList.add(taskNode.flowsIntoTask)
    }
    // for each task that this node waits for
    // add this to the current task
    for each (taskNode.waitsForTask) {
        currentTask.addWaitsForTask(taskNode.waitsForTask)
        // if this task is not already in taskList add it
        if (not taskList.contains(taskNode.waitsForTask))
            taskList.add(taskNode.waitsForTask)
    }
    // add the current task to the taskList
    taskList.add(currentTask)
}
```

Figure 33 Pseudo-code for the creation of task classes
Chapter 5: Aggregation

The pseudo-code in Figure 33 shows the process for gathering the information from the YAWL workflow plan. At the end of the process all of the tasks have been added to the taskList structure and these can then be used to translate the workflow.

Figure 34 shows XML exported by YAWL for the holiday booking service. Note that only the code between the <processControlElements> tags is needed to create a set of task classes and only one task class is shown for clarity. The important elements of this XML are as follows:

- Each task (including input and output condition) is defined by an opening and closing tag
- Between the tags is the information to determine the tasks that a task flows into. This information is used to construct the output algorithm
- By storing a list of tasks as processing progresses the tasks that a task waits for can be deduced from tasks in the list. For example, since inputCondition flows into FlightBooking, FlightBooking must wait for inputCondition. This information is used to construct the input algorithm.

```xml
<processControlElements>
  <inputCondition id="0_InputCondition">
    <flowsInto>
      <nextElementRef id="4_Car_Rental_Booking" />
    </flowsInto>
    <flowsInto>
      <nextElementRef id="2_Hotel_Booking" />
    </flowsInto>
    <flowsInto>
      <nextElementRef id="3_Flight_Booking" />
    </flowsInto>
  </inputCondition>
  <task id="3_Flight_Booking">
    <flowsInto>
      <nextElementRef id="1_OutputCondition" />
    </flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="Flight_Booking" />
  </task>
</processControlElements>
```
At the end of this process a list will have been created that contains all tasks that make up the workflow. The holiday booking service depicted in Figure 32 (page 83) would yield a set of classes similar to that shown in Figure 35. Note that flowsInto and waitsFor contain a list of tasks.
5.2.4 Task class creation

All task classes created use a common framework as depicted in Figure 36. The JFrame class allows the task to be rendered visually for monitoring purposes. The BoardListener interface ensures that the abstract task implements the notify method; in this case to listen for the arrival of messages that are directed to this task. AbstractTask ensures that the task classes implement the inputCondition and outputCondition methods; these methods contain the logic to either wait for the correct sequence of messages from upstream tasks and to send the correct sequence messages to downstream tasks.

The goal of this class structure was to allow tasks to be easily assembled using the translation program. The tasks generated using this structure are GC-based and can execute autonomously waiting for messages, processing messages, contacting services and then sending messages to other tasks (cf interaction machine). The flow of control inside the tasks in order to achieve this autonomous behaviour is explained in the next section.
5.2.5 Internal operation of a task

When the task classes are executed they monitor message arrivals in the mediator. The task consumes messages addressed to it and generates messages to other tasks.

Figure 37 Example from hotel booking scenario

The example in Figure 37 shows a snippet from the holiday booking service in Figure 32. The state diagram in Figure 38 governs the events that occur during execution.

Figure 38 State diagram for a GC-based workflow task
As an example, the events occurring during execution of the holiday booking service are:

1. The Hotel and Flight booking tasks each wait for a message from the Start task. The wait-for-messages phase is governed by the conditions programmed into the input algorithm. In this case the tasks are only waiting for one message to arrive.

2. As messages are received the input algorithm checks the input condition. When the condition is satisfied, internal processing occurs which may involve accessing local or remotes services. This processing is carried out concurrently and asynchronously with respect to other tasks in the aggregated service.

3. When processing of Hotel and Flight bookings is complete the output algorithm determines which messages are to be sent to which tasks. In this case Hotel and Flight will each send one message to the End task.

4. The End task waits for one message to be received from Hotel booking and one message from Flight booking. The End task may carry out further processing if required.

5. Once the End task process is complete the aggregated service can either terminate or restart.

### 5.2.6 Executing the aggregation

During translation an aggregation manager is also created. The purpose of this component is to invoke all of the GC-based classes that represent the tasks in the aggregation. These tasks then operate in an autonomous manner without further involvement of the aggregation manager. The aggregated service is initiated upon receipt of a request message. Messages (also known as “tokens” in YAWL) are generated and consumed by each of the tasks as the aggregation progresses. The messages contain 1) information which is useful for the completion of the service (for example prices and times etc) and 2) information which is useful for the completion of the aggregation (for example the state of the sending task). The action of a task depends on the embedded logic in each task and the content of the incoming messages. The aggregation is complete when the output condition receives messages satisfying its input condition.
Chapter 5: Aggregation

5.2.7 Run-time service selection

During the execution of an aggregated service the elements of the workflow (the tasks) may call other GC-based or remote services. If a service is unavailable the progress of the workflow may be halted indefinitely waiting for a response. GC-based mediation can be used in this situation to provide availability management at run-time. Rather than accessing a service directly all calls are made to the mediation layer. Mediation could redirect a call from unavailable service to another similar service based on a list of pre-programmed options.

5.3 Implementing workflow patterns using GC

There are many currently available workflow languages and systems but there is little in the way of organisational theory and standard business process modelling concepts [17]. The work by Van der Aalst et al just cited identifies useful routing constructs and presents these as a set of workflow patterns that can be used as a basis for the discussion of workflow implementations regardless of language or system. The usefulness of a pattern based approach to software development was originally investigated by Gamma et al. [59] and a pattern based approach to the design and composition of e-services has also been proposed [60].

In section 5.2.3, we discussed the translation of a simple workflow into a set of GC-based classes that could be used to execute an aggregated service. During the course of this research a number of workflow patterns were designed in YAWL, exported and then translated into a set of GC-based classes. These more straightforward cases which can be designed in YAWL are covered in appendix section 8.1. The more complex patterns, covered in the Appendix in section 8.2, have not been tested in YAWL. The purpose of the pseudo-code fragments presented in all patterns is to give an example of how each pattern could be implemented in a space-based language such as AOS.
5.4 Using GC based mediation as a distributed workflow engine

One of the main purposes of a workflow engine is to coordinate the flow of control between the tasks that make up the workflow. Generally, workflow engines coordinate workflow activities in a centralised manner, that is, the control flow logic is centrally located and managed. However, in a GC based environment the logic is distributed amongst the autonomous tasks which represent the workflow. Another distinction that can be made is in terms of where the coordination primitives are to be found. In an exogenous system such as Manifold [18] the coordination primitives are found outside of the entity that is to be coordinated. On the other hand, a GC-based system is endogenous, and the coordination primitives are found within the entities to be coordinated. In the case of this work, the endogenous aspect of GC allows workflow task classes to be created as discrete and complete entities containing all of the coordination and combinational logic to carry out the assigned task. If tasks need to be added or an existing task changed, the task is re-translated and then generated into the coordination space where they immediately become active. While an exogenous system may provide benefits due to the centralisation of the coordination code the endogenous model may lead to benefits of scale where tasks are run on different servers or maintenance benefits where tasks can be easily replaced without disruption.

There are several issues associated with workflow which are traditionally difficult to handle. This is also the case in the AOS environment. One problem, which has been under consideration by the workflow community [58] is related to the synchronising merge [17], also known as the OR Join. The problem with the OR Join is deciding when all of the incoming messages that are expected to arrive have either arrived or are not going to arrive. The latter is obviously the difficult part. One approach to this problem has been to treat OR-Joins as either an XOR –Join (pessimistic) or an AND-Join (optimistic) but this approach ignores the fact that the OR-Join can be both at the same time. Another approach is to ensure that all paths leading to an OR-Join are traversed as is the case with IBM MQSeries/Workflow where either a true or a false message is sent to each input and InConcert where a message is sent along all paths.
Chapter 5: Aggregation

Our approach to this problem is similar to that used by InConcert [17] but differs in that we use GC-based messages as tokens. That is, a message must be sent along all potential message paths that lead to an OR-Join. An OR-Join can expect to receive a message at all inputs and must wait until all messages arrive. A message can be a valid message or a null message; a valid message can contain any data relevant to the decision to be made by the OR-Join while a null message is simply a marker that indicates to the OR-Join not to expect a valid message from this path. The OR-Join conditional logic must take into account all messages (valid or null).

In Figure 39, an example scenario is depicted where task D is an OR Join. The solid lines show message paths and the dashed lines show paths where no valid message is generated. Tasks B and C represent, for example, XOR split style tasks where only one message is generated. Task B and C are aware that potentially a message could be sent to task D as this information has already been programmed into them. However, in this case, Task D would wait indefinitely and if Task E was an AND Join then it too would wait indefinitely for an incoming message.

With our approach, two types of messages are generated for all outgoing paths. Firstly, a valid message which contains information for another task (Figure 39, Task A produces two valid messages) and secondly, a “null” message which indicates to the downstream task not to expect a valid message (Figure 39, task B and C generate one valid and one null message each). The effect of this is that when task D receives two null messages it has enough information to continue processing and in Figure 39, task D will generate a null message which must be handled by task E. In a GC-based environment the state of the message (valid or null) can simply be indicated by the value of an internal attribute.
The use of null messages helps prevent cases of starvation and deadlock occurring in the workflow [17] by signifying that the traversal of a path was possible, but not exercised. However, if cycles are introduced into the workflow then cyclic null messages will have to be handled correctly by the tasks. One way of doing this is to record a history of tasks visited within the message as it progresses through the workflow. This would allow a task to know if the incoming message had been received before.

5.4.1 Coordination monitor program

During the research on aggregated services various scenarios arose which required prototyping and testing for validity. This led to the development of a test program (Figure 40) in which the scenarios were loaded and tested. The following scenarios were programmed:

![Coordination Monitor](image)

Figure 40 Scenarios were tested using a purpose built program
Chapter 5: Aggregation

**OR-Join**

This option creates a set of tasks that rely on an OR-Join for the completion of an aggregated service. This test uses the “null token” method (cf. InConcert [17]) so that all paths in the workflow are traversed even if there would not normally have been a message sent along a particular path. During execution of the test random paths are exercised, that is, some paths may be exercised with valid messages and other paths are exercised with “null” messages. The purpose of the null message is to indicate to the downstream task that no valid message will be sent and it can make an informed decision about what action to take.

**Deadlock**

One example of deadlock is when a task is waiting for a message which either directly or indirectly emanates from itself. One cause of deadlock is faulty design. At run-time this is a very difficult situation to deal with and the test carried out was more to do with trying to detect if a workflow contained potential deadlocks. This is achieved by creating a message that records the tasks that it passes through as it “travels” through the workflow. If a task detects that a message has passed through it before then this is a cyclic path and there is the potential for deadlock to occur. At this point the task can generate a cancellation message (see below) and terminate the test.

**Notify**

This option was to test the ability to change the behaviour of a task while it is executing. This can be achieved by sending a message to the task to reconfigure its notification at run time. When the message is received the task deregisters notification for one or more class of messages and adds notification for a new class or classes of messages. Thus the behaviour of the task can be changed at runtime without withdrawing the task. One issue, which results from the use of design tools, is that of brittleness. When a design has been completed and the aggregate service implemented, any change in the services upon which the workflow relies may cause the workflow to cease to function. This is known as brittleness and usually results in the workflow having to be redesigned and redeployed.

In a GC-based system tasks can be replaced without the need to redeploy the entire aggregate service. This can be achieved because the tasks operate in an autonomous
manner and are decoupled from each other by the nature of GC. In a typical aggregate service scenario, if one of the components of the service needs to be rewritten to change the internal logic the task can be cancelled and then immediately replaced. Provided that the replacement occurred while that part of the aggregate service was not active the replacement could be made without disruption.

**Distributed behaviour**

In chapters 3 and 4 we discussed the passing of behaviour between services and clients. This function tests the ability to embed behaviour in a message and then pass the behaviour to a recipient for execution.

**Multi Path**

This test scenario was used to investigate the use of null messages to ensure that a task that was simulating an OR Join would be able to handle a varying number of indeterminate inputs. For example, an OR-Join waiting for three inputs could receive a combination of valid or null messages. The OR-Join must be capable of handling all cases in order for the workflow to proceed normally. This test was used to simulate the behaviour of the OR-Join upon receipt of random combinations of valid and null messages.

**Cancellation of workflows**

This test was used to test simple cancellation of a task or of a single workflow instance. All tasks can listen for one or more cancellation messages. Therefore, a cancellation message can be “addressed” to a single task, the tasks that form part of a workflow or to all tasks in a workflow. When a task receives a cancellation message, its execution is terminated.

**5.4.2 Summary**

In this chapter, we looked at using GC based mediation to create aggregated services based on a workflow plan and proposed a four-stage process. First, a design of an aggregated service is created using a workflow editor such as YAWL for example. Second, this design is translated to a set of AOS Active Objects that interact with each other using GC and coordinate the intention of the workflow. Third, the aggregated
service is executed and managed by GC-based mediation and finally the aggregated service is invoked by a client and performs the design service.

Also in this chapter we examined a set of design patterns which have been proposed by Van der Aalst et. al. [17] and showed how these could be implemented using GC in an AOS environment (see appendix 8.1 and 8.2).

In the next chapter, we combine the elements discussed in the last three chapters and propose a service delivery environment that can be used to design, deploy and manage services. In particular, we will focus on the design of aggregated services, the creation of GC-based services, the re-use of existing services and the invocation of services by clients using a generic protocol.
Chapter 6

6 A proposed GC based environment for services

In chapters 3, 4 and 5 we proposed GC as an alternative communication medium for client/service interactions. We showed how a GC-based layer of mediation could be introduced between clients and services to provide, to a limited extent, facilities such as aspects of heterogeneity resolution, adaptation, capacity management and aggregation of services. In this chapter, we draw on the research carried out in the previous chapters to propose a GC-based service environment. While this is a difficult and ambitious goal the environment presented here has its merits and goes some way to providing facilities that can be used by service developers and service users. Some of the elements of the service environment proposed in this chapter were created as mock-ups or prototypes for the testing of various aspects of previous chapters.

A major goal of the service environment is to provide a holistic approach to the development, aggregation, publishing and invocation of services. In doing this we will firstly outline the architecture from the perspective of the main architectural components of mediation, aggregation and services and then take an orthogonal role-based view and present our environment from two perspectives: firstly, how service providers can publish several types of services; GC-based services, location services, services with heterogeneous protocols and aggregated services and secondly, we will take the view of service users that search for services and invoke services.

Therefore, the ultimate goal of such a service environment would be to provide a unified approach to the publishing, finding and invocation of services. There are two main aspects to this unified approach. Clients always interact with the service environment using a common GC-based protocol which may be adapted to provide a façade to other protocols. Similarly, services are published in a consistent manner using a standard GC-based protocol regardless of the language the service is written in (eg. C++, CGI, Java etc) and regardless of the protocol the service uses (eg. SOAP etc).
As mentioned above, one aspect of this research is to view the proposed service environment from a role-based perspective. That is, from the perspective of service developers, service locators, service users and aggregated service developers. Figure 41 shows how the various roles of the service environment relate and what activities are performed by the roles.

Figure 41 Interaction of roles with the service environment

6.1 Service environment architecture

Figure 42 Service environment architecture components
The service environment can be viewed as three distinct conceptual components: mediation, aggregation and services. These are all supported by GC and also by extended features such as AOS notification (Figure 42). The main role of mediation is to intercede between service participants during interactions to provide facilities that leverage the underlying communication mechanism. Mediation is the first point of contact for service users and service providers. Service users send requests for services to mediation and these requests may either be passed directly to the required service (in the case of a GC-based service) or adapted (in the case of remote services). Mediation based adaptation also provides the opportunity to resolve issues of heterogeneity among similar services that have differences in their interfaces. Service providers (developers) create either GC-based services or remote services. GC-based services execute within the service environment and can monitor and interact with the service environment. Remote services can either be new services using existing protocols or reuse of existing services that use existing protocols (eg. SOAP, CGI, etc).

The main benefit of mediation investigated in this research is backward compatibility to existing services that use a variety of protocols – clients use a common protocol to connect various services and mediation provides adaptation to heterogeneous protocols. Other benefits mediation can provide are: 1) Availability management - mediation redirects calls to other services when services are unavailable, 2) Capacity management – mediation adds additional services to alleviate short-term demand for services.

The aggregation component manages the provision of aggregated services within the service environment. The design of the aggregated service is carried out using a workflow designer such as the YAWL editor. Workflow designs are then translated into a GC-based implementation and registered with a location service. From the user perspective, aggregated services are found and invoked in the same way as other services. The service environment provides access to four types of services:

Remote services
Remote services are services outside of the GC environment that use non-GC protocols. Service developers create new remote services or re-use existing remote services. These services use a variety of protocols such as SOAP and CGI for example
GC-based services

These are services that are implemented using a GC-based language such as Active Object Space (AOS) and are capable of interacting directly with GC-based mediation. GC-based services can be implemented at a remote location or can be created remotely and sent to the service environment for execution. GC-based services can be used to provide access to physical services such as printers, faxes and spell checkers (see Chapter 4 Interaction mediation).

Location services

Location services are a specialised type of GC-based service (similar to UDDI registries) that allows service developers to register services and their descriptions and also allows service users to find registered services by matching elements of the service descriptions. Location services advertise their presence by generating query objects (for users) and description objects (for developers) into the space.

Aggregated services

Aggregated services are GC-based services that developers create and publish through the aggregation component. Aggregated services are implemented as a set of GC-based active objects that represent the operations and coordination of the workflow from which the aggregate service was designed.

Distributed behaviour

The service environment leverages the ability of AOS to pass executable code between clients and services. This ability is used to decouple service interfaces by passing GUI code that encapsulates the interface as part of the interaction. This means that clients do not need to know the details of the service interface but simply invoke the graphical representation of the interface distributed by the service. The user then manually interacts with the GUI. Alternatively, as mentioned in Chapter 5, if a direct GUI interaction with the service is not appropriate the service could pass either 1) a program which generates local “stub” code for the client programmer to use, or 2) a Java object which can be used as an API for client-side Java code using reflection.
techniques. In this research, only the GUI invocation interface is considered because this reduces the amount of client coding required and is easier to demonstrate.

6.1.1 Initialising the service environment

When the service environment is started, its associated space would not contain any messages or objects. In order to assist users and developers, a set of specialised tools or services can be made available from the service environment at start-up and maintained while it is active. These tools could include:

Initial location service
An initial location service can be provided by the service environment so that any developers wishing to register a service were able to do so. Developers could publish additional location services later but an initial location service would be of benefit for bootstrapping purposes.

Location service locator (LSL)
A location service locator can be used to allow a user to view all currently active location services and to select one of them for querying. It would also be possible for a user to find location service query objects manually but the location service locator automates this process. Since different location services would have different query interfaces, queries cannot easily be sent to more than one location service at a time automatically.

Service publisher
The service publisher program, as its name implies, assists service developers in publishing services. Publishing services is a matter of finding location service description objects from available location services and registering the service description with the desired location service. The service publisher eases this process by presenting a list of available location services (similar to the location service locator) and guiding the developer through the registration process.
6.1.2 Service environment prototype

A mock-up of the proposed service environment is shown in Figure 46 (page 104). In a typical scenario, at least one “bootstrap” location service would have to be published to allow service providers to publish services and for service users to find services. Aspects of using the proposed service environment prototype are now summarized to give an overview of the main processes.

Service creation and publishing: Firstly, service providers lodge details of their service into the service environment using the Service Publisher GUI (Figure 43). These details allow the mediator to create an adapter to make the service available to clients using the common GC-based protocol. Secondly, service providers must also create an invocation GUI for registration with a location service along with the description of their service. Both of these operations are carried out at the same time.

Service searchers: Use the Location Service Locator to first find any location services that have been published (Location services you will recall are similar to UDDI registries). Once a location service is selected (eg. FastFind) the client can
search for registered services. For example (Figure 44) the Fast Find location service can be used to locate a particular printer service.

![Figure 44 Service locator and service finder](image)

**Service invokers:** Fill in the invocation GUI (passed from the service publisher to the location service) and invoke the service (Figure 45).

![Figure 45 Service invoker GUI](image)

The overall view of the proposed service environment mock-up/prototype is shown in Figure 46.
In the next section, we look at how a service environment based on the architecture just presented can be viewed from the perspective of the roles of service publishers and service users.

Figure 46 The service environment mockup
6.2 Service environment roles

The use of the service environment can also be viewed from the perspective of the roles that a potential user will assume and the tasks that the user can perform in a particular role.

As can be seen from Figure 47 most of the interaction with the service environment occurs with the mediation component. These roles are now examined in more detail.

6.2.1 The service developer role

Service developers can create three types of services and then deploy these services into the service environment using the service publisher tool:

**GC-based services**: These are services that are designed a priori to use GC as the sole means of client/service interaction. GC-based services as proposed in this research, are used to make local services such as hardware (printers etc) and software (eg. spell...
checkers) available to clients of a GC-based environment. The developer of a GC-based service may be looking to develop services that are available on a local basis, for example within a corporate intranet. A good example of this would be printer services that represent multiple printers of varying types available in different buildings and on different floors of a building.

**Remote services:** These are services that use one of a variety of existing protocols (eg. SOAP, CGI etc). Remote services represent either existing services that may have been in service for some time or new services that are being developed. Remote services can be developed for their intended protocol where the main effort is expended and then deployed to a GC-based environment with minimal extra effort by coding an adapter. Reusing services in this way increases their availability without a large investment in re-deployment.

**Aggregated services:** These are services composed of other services and are developed from a workflow plan. In this research aggregated services are composed of tasks which carry out the coordination required by the aggregated service. At various stages during the execution of the aggregated service other services (GC-based or otherwise) can be contacted to perform processing or gather data as dictated by the design of the aggregated service.

**Publishing Services**

Once a service has been created the developer must retrieve a service publisher program (Figure 48) from a location service. The service publisher allows the developer to define the service protocol, location and parameters and to send the service description back to its location service for registration. Note also the inclusion of a class file which represents the invocation GUI for the service. By comparison, in a SOAP environment the service developer would need to contact the operators of a service registry and enter into an agreement for the publishing of the web service. This would normally be done using a commercial web site (eg. www.strikeiron.com).
Sequence of events is as follows:

1. Retrieve service publisher program
2. Enter the description of the service and send to the location service
3. The mediator intercepts the description and creates an adapter

**Location service developer role**

As mentioned in Chapter 4 (Interaction mediation), while UDDI based registries are a part of the web service architecture, location services are not a separate part of the GC-based architecture and are implemented the same as other services. This means that location services share the same status as other services in the service environment. Therefore, the same protocols are used to interact with location services as are used to interact with other services. However, making location services available is somewhat different to other services since these services are not normally registered with a location service (but this is not precluded). Location services are
Chapter 6: A GC-based environment for services

implemented as AOS Active Objects as are other GC-based services such as those mentioned earlier in this section.

Aggregate service developer role

The design and development of aggregate services can be carried out local to the service environment or at a remote location. The result of the workflow design process (XML) must be sent to the service environment for translation and invocation. Translation is a matter of reading the design specified in the XML and creating a GC-based set of Java classes that represent the aggregated service.

Figure 49 Aggregated service development

Aggregate service developers design aggregate services using a workflow editor such as that provided by YAWL (Figure 50).

Figure 50 Example of simple YAWL workflow design tool

When complete, the design is translated into a set of GC based Java classes using the workflow translator tool (Figure 51). If this process of designing an aggregated
service in YAWL was carried out in a SOAP environment the process would be somewhat different. One possible SOAP implementation would be to use the content of the YAWL XML to create a “hard-coded” control program (cf. Exogenous) that contains the interaction logic specified in the YAWL diagram. The control program would also be hard-code to contact any remote web services specified in the workflow, as the workflow progressed.

![Workflow Translator](image)

Figure 51 Workflow translator tool GUI

The purpose of the aggregate service manager (Figure 52) is to initiate the aggregated service by sending a message to the “start element”. When the aggregated service is complete and the terminator task is reached control is passed back to the aggregate service manager. The aggregate service manager GUI shows a monitor window for each task in the aggregated service. This is used to monitor the progress of the service.
6.2.2 The service user role

Once some services have been developed and published, potential clients are free to search for and invoke services. These roles are now discussed in more detail.

The service searcher role

The first interaction that a user has with the service environment would most likely be to search for a suitable service. Since the proposed service environment can support multiple location services the user can either use a known location service or retrieve a location service locator (LSL) program (Figure 53) from the mediator. The LSL shows all of the location services that are currently active in the service environment and allows the user to select one to query.
Once a suitable location service is found, the user interacts directly with that location service. For example, if the Fast Find location service is selected, the LSL would retrieve the query interface for the Fast Find location service (Figure 54) from the mediator. The user can then enter the details of the required service and if a matching service is registered at Fast Find the interface for the service would be returned and the searching process is complete. The next phase of operation would be as a service user.

The process the user would follow to locate a service is as follows:

1. User retrieves a query object for a particular location service
2. Enters details of the desired service and sends the query object back to the location service
3. The location service receives the query object and finds a matching service
4. A service object is returned to the user. The service object is then used to invoke the service

While the above process allows clients to select a specific location service and to address a query to the location service, an alternative concept would be to send a generic query to the mediator. This would allow the query to be “pulled” by any interested location service. This type of generic interaction is not naturally available in a SOAP based UDDI environment where interactions between a prospective client and a registry are based on the query protocol adopted by that registry provider.

**The service user (invoker) role**

When the service description was published to the location service it also contained an invocation interface. When a service is selected from the location service this invocation interface is returned with the service details and invoked using the service invoker (Figure 55). The GUI passed from the service is the Service Parameters panel.

![Example GUI for service invoker](image)

Figure 55 Example GUI for service invoker

Depending on the service selected, different service parameters may be displayed. The service parameters panel is defined by the service and is passed to the location service during registration and on to the client when the service is selected. This is an
example of distributed behaviour mentioned in chapters 3 and 4. A response may be returned at the end of the service invocation and displayed.

The sequence of events is as follows:

1. Service user retrieves a service description object and the invocation interface is displayed in the service invoker
2. User adds parameters and invokes the service
3. Service returns a result object to the user

6.2.3 Summary

Current service delivery environments such as for example web services, CGI and CORBA all use their own protocols for client/service interactions. This means that providing the same (or similar) services under multiple protocols requires a duplication of effort at the service end. Also, a client program developer who wishes to use services from all of these protocols would have to master the subtleties of each service environment which leads to an increase in effort at the client end. If a one-protocol-fits-all approach could be achieved this would lead to a number of benefits. Firstly, a reduction in complexity at both the client end and the service end as just mentioned, secondly the ability to re-use existing services and finally, the ability to increase the range of availability of a service outside of its native protocol.

Therefore, one goal of this service environment is to propose a unified means of accessing existing services that use disparate protocols. As presented in chapters 3, 4 and 5 this is achieved by providing a GC-based layer of mediation that intercedes between clients and services to adapt the protocol of the target services to a common
GC-based protocol. This has two benefits, firstly services that use different protocols (eg. SOAP, CGI etc) can be accessed using a single protocol and the developer need only learn one protocol and secondly, the single protocol approach opens up a wide range of existing services, promotes reuse and introduces backward compatibility. While this approach has challenges of its own, it avoids having to repeatedly “re-solve” the challenges faced by individual service environments during their development.

However, the main goal of the service environment is to propose an integrated toolset that can be used by all parties in service interactions. Three main roles emerge when considering such a service environment: 1) Service developers are able to create new services or reuse existing services (regardless of the protocol) using publishing tools while aggregated service developers can create and launch aggregated services based on workflow plans. 2) Service users can search for suitable services by using the location-service locator tools and then find services using registered location-services. 3) Once the desired service is found service users can then invoke it. A side-benefit of using the AOS for the service environment is that whenever interaction with a service (either a location service or actual service) is required during these phases, distributed behaviour can be used by passing GUI representations of the service interface as executable code.

While the main strengths of GC are the decoupling it provides at the address level and by decoupling interactions in time, another aspect that is evident during the development of a service environment prototype is that because GC has only a small set of operations or primitives, these are easily learned and the developer need only consider three main concepts of interacting entities and their messages; that is Notify, Generate and Consume. Arbab [18] (a supporter of the centralised approach to coordination) is of the opinion that having GC primitives interspersed with code makes the cooperation model and the coordination protocol “nebulous and implicit”. However, the benefit of using the GC paradigm is that the coordination code can be placed where it is needed and where it makes sense; that is, inside the entity that is being coordinated, thus creating examples of Wegner’s interaction machines. This approach is also similar to encapsulation, one of the corner stones of object oriented programming.
Chapter 7

7 Conclusions and future work

In the chapters 3, 4 and 5 we proposed GC as an alternative communication medium for a service environment that facilitates mediation and aggregation while in Chapter 6 we proposed a prototype service environment and a set of tools to facilitate its use. In this chapter, we conclude the thesis by summarising the advantages and disadvantages of our approach and proposing future work.

7.1 Summary and conclusion

RPC communications, as used by most web services, were originally designed to aid programming by providing remote interactions with local semantics. However, the local semantics of RPC gives rise to the naïve assumption that the network, which connects one machine to another behaves transparently, but consequently due to network latency and outages, this is rarely the case. While RPC communications aim for simplicity at the programming level they have a number of disadvantages, they are usually synchronous and usually require static, pre-known network addresses to establish communications.

Generative Communications is an alternative to RPC communications which provides two levels of decoupling not found in RPC based communications: 1) by buffering messages between senders and receivers temporal decoupling is introduced, and 2) by using associative addressing spatial decoupling is introduced and static network addresses can be avoided. While it could be argued that GC is a more complicated means of network interaction than RPC, most GC implementations have only a small set of primitive operations and any increase in complexity should be balanced against the asynchrony and decoupled addressing that is naturally provided in a GC based environment.
Mediation
In this research, we proposed GC based mediation as a layer of middleware to intercede between clients and services providing additional features. There are two main advantages of this approach that emerge from the research. Firstly, mediation based adaptation can be used to resolve issues of heterogeneity between services that have a similar purpose but have different interfaces and secondly, mediation based adaptation can also be used to allow interaction with services that use different protocols such as for example CGI or SOAP. This second aspect introduces the possibility of re-using existing services from a variety of protocols therefore providing a degree of backward compatibility. The developer of clients for these “mediated” services only needs to learn how to use the mediation environment protocol and does not need to learn the intricacies of a suite of disparate protocols. The adaptation to the individual services and their protocols is done once by mediation and reused whenever the service is invoked.

The disadvantages of a GC based environment are mostly to be found in the overheads that are required to support this form of mediation. Firstly, because GC relies on buffering of messages during the generate/consume process there will be a degree of latency involved which would be proportional to the number of messages being handled by the mediation environment at any given time. Secondly, each of the adaptors that are required to connect to the remote services would have to be developed at a cost of time and materials. However, as mentioned above, this would be a one-off cost which could be recouped if increasing numbers of users invoked the service.

Location services
In Chapter 4, location services were introduced as an alternative (and extension) to UDDI registries. Location services are implemented as GC-based services that provide facilities for service providers to register services and for service users to find services. However, while UDDI registries are a part of the web services architecture, location services are the same as any other GC-based services
Chapter 7: Conclusions and future work

**Aggregation**

We also examined the aggregation of simple services into more complex ensembles based on a workflow plan. We outlined how a workflow plan generated in the YAWL editor could be translated into a set of GC based autonomous classes which implemented the structure and logic of the workflow and used GC as an interaction medium. Current workflow implementations can some times exhibit an attribute called brittleness when the failure of one part of the workflow causes the entire workflow to fail. GC based aggregation overcomes this problem to some extent by allowing the possibility for tasks to be dynamically swapped-out at run-time if they fail. Also, when a workflow needs to be modified it must be redesigned, regenerated and re-implemented. In a GC based aggregation, because there is no static address-based connection between tasks, elements of the workflow can be re-implemented if required even while the workflow is still active.

### 7.2 Future Work

There are a number of directions future work could take and during this research topics such as scalability, security and availability management could have been examined. However, each of these topics would involve much challenging research in its own right. A longer-term aim would be to integrate all parts of this work (and some not yet considered) into a working version of the service environment based on that presented in Chapter 6.

There also some questions as to the quality of service (QoS) that could be provided by GC based system of interaction. Generally, if a web service is busy any request will block until the service becomes available. Due to the decoupled nature of GC once the message has been generated the client is free to carry out other operations, return later to check for a response. However, if there are a large number of messages being generated, some delays could be experienced and QoS could be a further topic of research.
8 Appendix 1: Workflow patterns

In this appendix, we have taken the workflow patterns proposed by van der Aalst [17] and applied GC programming techniques to create pseudo-code that gives an indication of how the pattern can be implemented in a GC environment. The appendix is divided into two sections. Firstly, basic control flow patterns which implement a single workflow element such as an OR join and secondly, complex control flow patterns which involve more than one workflow element. Flash animations of these patterns are available at the web site shown below. The reader is encouraged to view these animations to gain a better understanding of the purpose of the workflow pattern. http://is.tm.tue.nl/research/patterns/flashAnimations.htm

Terminology used in the pseudo-code

**Message**: The term message is used in the following text to indicate a unit of communication that flows between tasks. In the YAWL literature, these are referred to as tokens but we have chosen here to use the term message which has been used in previous chapters.

**Register**: If a task is to be notified when a message arrives it must register with AOS by using a template that specifies the format of matching messages. A template is a java class that allows messages to be matched by class name or by the values of public attributes.

**Notify**: This is used to represent the toDo method in AOS. This is a call-back method which receives a message matching the template that was registered. When a message is passed to the notify method it can either be “read” (not removed) from AOS or “taken” (removed) from AOS. In the case of the patterns presented here all messages are taken.

The example pseudo-code shown in the next two sections is to demonstrate one approach that may be used to implement the workflow patterns using GC. The
Appendix 1: Workflow patterns

Examples do not take into account complex issues such as cycles, deadlocks or starvation that may occur during execution.

8.1 Basic control flow patterns

In this section we examine in some detail the basic workflow patterns proposed by van der Aalst [17]. Each pattern is presented with a brief description, an example of the YAWL symbol (if available), an example task diagram showing the relationship of the elements in the pattern, an example of the YAWL generated XML (if available) and a pseudo-code example of how the pattern might be implemented in a GC based environment.

Note: the XML presented does not include standard XML headers and footers which are common to all XML generated by YAWL.

Pattern 1: Sequence

This pattern is the simplest and involves the passing of a message from one task to another, to another etc. in a sequence. When a message is consumed by the task an activity may take place within the task before a message is generated to the next task.

XML generated by the YAWL editor for this pattern is:

```xml
<specification uri="Pattern1.ywl">
  <metaData />
  <schema xmlns="http://www.w3.org/2001/XMLSchema" />
  <decomposition id="New_Net_1" isRootNet="true" xsi:type="NetFactsType">
    <processControlElements>
      <inputCondition id="0_InputCondition">
        <flowsInto><nextElementRef id="3_TaskA" /></flowsInto>
      </inputCondition>
      <task id="3_TaskA">
        <flowsInto><nextElementRef id="4_TaskB" /></flowsInto>
        <join code="xor" />
        <split code="and" />
        <decomposesTo id="TaskA" />
      </task>
    </processControlElements>
  </decomposition>
</specification>
```
Each class that is generated from the XML to simulate a task is implemented as an AOS active object (ie. an object that executes in its own thread of control).

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class TaskA extends ActiveObject

    Register(messageA)

    ... 

    method Notify(message)
        process(message)
        messageB = contactWebService()
        write(new messageB)
```

The AOS implementation uses a task which waits for a single message, carries out some process and then writes another message addressed to the next task in the sequence.
**Pattern 2: Parallel Split (AND Split)**

Parallel split allows a single thread of control to be split into multiple threads of control that can execute concurrently. The YAWL symbol for this pattern and a typical application are:

![Parallel Split Diagram]

XML generated by the YAWL editor for this pattern is:

```xml
<processControlElements>
  <inputCondition id="0_InputCondition">
    <flowsInto><nextElementRef id="4_AndSplit" /></flowsInto>
  </inputCondition>
  <task id="4_AndSplit">
    <flowsInto><nextElementRef id="3_TaskA" /></flowsInto>
    <flowsInto><nextElementRef id="2_TaskB" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="AndSplit" />
  </task>
  <task id="3_TaskA">
    <flowsInto><nextElementRef id="1_OutputCondition" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="TaskA" />
  </task>
  <task id="2_TaskB">
    <flowsInto><nextElementRef id="1_OutputCondition" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="TaskB" />
  </task>
  <outputCondition id="1_OutputCondition" />
</processControlElements>
```

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class ParallelSplitTask extends ActiveObject

    Register(messageA)
```

121
Appendix 1: Workflow patterns

... method Notify(message)
    process(message) // carry out some process
    write(new messageB)
    write(new messageC)

The AOS implementation shows a parallel split task that waits for message A, then carries out a process before writing two new messages. This pattern allows a process to be split into multiple concurrent downstream tasks.

Pattern 3: Synchronisation (AND Join)

Synchronisation allows multiple threads of control to merge into a single thread of control. The YAWL symbol for the and-join and a typical application is:

XML generated by the YAWL editor for this pattern is:

```xml
<processControlElements>
  <inputCondition id="0_InputCondition">
    <flowsInto><nextElementRef id="2_TaskB" /></flowsInto>
    <flowsInto><nextElementRef id="3_TaskA" /></flowsInto>
  </inputCondition>
  <task id="3_TaskA">
    <flowsInto><nextElementRef id="4_AndJoin" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="TaskA" />
  </task>
  <task id="2_TaskB">
    <flowsInto><nextElementRef id="4_AndJoin" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="TaskB" />
  </task>
  <task id="4_AndJoin">
    <flowsInto><nextElementRef id="1_OutputCondition" /></flowsInto>
    <join code="and" />
    <split code="and" />
    <decomposesTo id="AndJoin" />
  </task>
</processControlElements>
```
Appendix 1: Workflow patterns

Typical pseudo-code to implement this pattern in a GC environment is:

```java
class Synchronisation extends ActiveObject
{
    register(MessageA)
    register(MessageB)
    ...
    method notify(message)
    {
        if (not flagA and message = messageA) flagA = true
        if (not flagB and message = messageB) flagB = true
        if (flagA and flagB)
            process(messageA,messageB)
            write(new messageC)
    }
}
```

The Synchronisation task waits for a specific number of incoming messages to arrive before carrying out some process and finally generating an outgoing message.

**Pattern 4: Exclusive Choice (XOR Split)**

Exclusive choice transfers control to only one of a number of possible choices. The YAWL symbol for the and-join and a typical application are:

![Diagram of XOR-split task and And Split]

XML generated by the YAWL editor for this pattern is:

```xml
<processControlElements>
    <inputCondition id="0_InputCondition" />
    <task id="4_TaskB">
        <join code="xor" />
    </task>
</processControlElements>
```
Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class ExclusiveChoice extends ActiveObject

    Register(messageA)
    ...

    method notify(message)
        if (message == messageA)
            if (someDecision = true)
                write(new messageB)
            else
                write(new messageC)
```

When the incoming message arrives Exclusive Choice generates only one of a number of possible output messages based on a decision.
Appendix 1: Workflow patterns

**Pattern 5: Simple Merge (XOR Join)**

Multiple paths merge into a single path but it is assumed that only one path is executed at a time (ie. there is parallel execution of multiple input paths). The YAWL symbol for the and-join and a typical application are:

![Diagram of XOR Join task]

XML generated by the YAWL editor for this pattern is:

```xml
<processControlElements>
  <inputCondition id="0_InputCondition" />
  <task id="3_TaskB">
    <flowsInto><nextElementRef id="2_XORJoin" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="TaskB" />
  </task>
  <task id="2_XORJoin">
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="XORJoin" />
  </task>
  <task id="4_TaskA">
    <flowsInto><nextElementRef id="2_XORJoin" /></flowsInto>
    <join code="xor" />
    <split code="and" />
    <decomposesTo id="TaskA" />
  </task>
  <outputCondition id="1_OutputCondition" />
</processControlElements>
```

Typical pseudo-code to implement this pattern in a GC environment is:

```java
class SimpleMerge extends ActiveObject

    Register(messageA,messageB)
    ...

    method notify(message)
        DeRegister(messageA)
```

125
DeRegister(messageB)
if (message == messageA)
    write(new messageC)
    Register(messageB)
Else // messageB
    write(new messageC)
    Register(messageA)

SimpleMerge waits for a number of messages. When one arrives, SimpleMerge ceases notification for the other messages and handles the incoming message. Once the message is handled SimpleMerge re-registers for the remaining messages.

**Pattern 6 : Multi Choice (OR Split)**

A single path is split into one or more paths. The YAWL symbol for the and-join and a typical application are:

XML generated by the YAWL editor for this pattern is:

```xml
<processControlElements>
    <inputCondition id="0_InputCondition" />
    <task id="3_TaskB">
        <join code="xor" />
        <split code="and" />
        <decomposesTo id="TaskB" />
    </task>
    <task id="4_TaskA">
        <join code="xor" />
        <split code="and" />
        <decomposesTo id="TaskA" />
    </task>
    <task id="2_ORSplit">
        <flowsInto>
            <nextElementRef id="3_TaskB" />
            <predicate>true()</predicate>
            <isDefaultFlow />
        </flowsInto>
    </flowsInto>
</processControlElements>
```
Typical pseudo-code to implement this pattern in a GC environment is:

```java
class MultiChoice extends ActiveObject

    Register(messageA)
    ...
    method notify(message)

        if (decisionA)
            write(new messageB)
        else if (decisionB)
            write(new messageC)
        else if (decisionC)
            write(new messageB)
            write(new messageC)
```

On receipt of an incoming message MultiChoice generates one or more outgoing messages based on an internal decision.

**Pattern 7: Synchronising Merge (OR Join)**

Multiple paths are merged into a single path. Any combination of incoming messages may be received.

Typical pseudo-code to implement this pattern in a GC environment is:
Class SynchronisingMerge extends ActiveObject

```
Register for messageA and messageB
...
method notify(message)
  if (message == messageA)
    flagA = true;
  else if (message == messageB)
    flagB = true;
  if (flagA and flagB)
    if (messageA.isNull())
      // some process
    else if (messageB.isNull())
      // some other process
    else
      // some other process
    write(new new messageC);
```

SynchronisingMerge expects two incoming messages to arrive. When both expected messages arrive some processing is carried out and an outgoing message is generated.

**Special conditions for the OR Join**

One problem with the OR-Join is "decidability" [58]. The OR-Join must receive a message from all "expected" incoming paths before a decision can be made and an output message generated. There may be cases where an incoming message is not generated by an upstream task due to the failure of a condition. In order for the OR-Join to decide a course of action (ie. decide if a message is expected or not) this failure must be communicated. One way of achieving this is to ensure that all incoming paths are traversed either by a valid message (ie. one containing say a true or false or some other value) or by a "null" message. The purpose of the null message is to indicate that the upstream task is complete but that no valid message was generated for the given path. This allows the OR-Join to decide what action to take; even if some part of the decision process relates to null messages that were received.
8.2 Complex control flow patterns

In this section we present workflow patterns that are more complex than those in the previous section. In most cases these patterns cannot be represented by a single task or component and they generally contain behavioural aspects that cannot be shown in a static diagram. Please refer to the flash animations for these patterns.

Pattern 8: Multi-Merge

Multi-merge merges multiple paths into a single path. Each incoming path that is activated results in an outgoing message even if multiple messages arrive concurrently.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class MultiMerge extends ActiveObject

    Register(messageA, messageB)
    ...
    method notify(message)
        messageC = someProcess(message);
        write(new messageC)
```

Pattern 9: Discriminator

Discriminator waits for a particular message to be received before rejecting further incoming messages and resetting. Only one message will be output.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class Discriminator extends ActiveObject

    Register(messageA, messageB)
    ...
    ignoreRest = false
```
...  

    method notify(message)  
    if (ignoreRest and allMessagesReceived)  
        reset()  
    else if (message == messageA)  
        processA()  
        write(new messageC)  
        ignoreRest = true  
    else if (message == messageB)  
        processB()  
        write(new messageC)  
        ignoreRest = true  

The discriminator waits for a predetermined number of incoming messages. When this has occurred all other incoming messages are ignored, processing takes place and an output message is created. When all incoming paths have been activated Discriminator resets.

**Pattern 10: Arbitrary Cycles**

Paths in a workflow can be repeated based on user input or a decision.

When message B is received at the XOR split, either message C is created or message D is created based on a decision.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class Merge extends ActiveObject

    Register(messageA,messageC)
```
...  
  method notify(message)  
    write(new messageB)

**Class XORSplit extends ActiveObject**

    Register(messageB)
    ...
    method notify(message)
      if (waitForInput(5000))
        write(new messageC)
      else
        write(new messageD)

The Merge task generates a message for XORSplit. Based on a user or machine decision either message C is generated back to the Merge or the process times-out after 5 seconds and message D is generated to downstream tasks.

**Pattern 11: Implicit Termination**

A sub-process terminates when there is nothing else to be done.

Typical pseudo-code to implement this pattern in a GC environment is:

**Class ImplicitTermination extends ActiveObject**

    Register(messageA)
    ...
    method notify(message)
      if (waitForSomeUserEvent())
        write(new messageB())
        terminate

A message arrives and after a user initiated event occurs a message is generated and the process terminates.
**Pattern 12 : Multiple Instances (without synchronisation)**

One or more other processes can be created that execute in their own thread of control. These processes are not synchronised and can carry out other ancillary tasks and terminate at their own discretion.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class MultipleInstances extends ActiveObject

    Register for messageA
    ...
    method notify(object)
        write(new ProcessA())
        write(new ProcessB())
        write(new ProcessC())
```

When a message is received one or more concurrent processes are started. In an AOS-based system these could be Active Objects.

**Pattern 13 : Multiple Instances (with a priori design time knowledge)**

At design time it is known that a process will trigger three sub-processes (A, B and C). When these three sub-processes are activated they wait for a decision to be made and when all three are complete the main process continues.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class MultipleInstances extends ActiveObject

    Register for messageX,messageA,messageB,messageC
    ...
    method notify(message)
        if (message==messageX)
```
write(new SubProcessA())
write(new SubProcessB())
write(new SubProcessC())
if (message==messageA)
    flagA = true
if (message ==messageB)
    flagB = true
if (message ==messageC)
    flagC = true
if (flagA and flagB and flagC)
    write(new messageD())

class ActiveObjectA extends ActiveObject
    Wait for a condition
    Write(new messageA)

class ActiveObjectB extends ActiveObject
    Wait for a condition
    Write(new messageB)

class ActiveObjectC extends ActiveObject
    Wait for a condition
    Write(new messageC)

In response to messageX, class MultipleInstances generates three Active Objects processes A, B and C which begin executing in their own thread of control. At some stage during the execution of these processes (perhaps in response to user input) messages A, B and C are generated. When all three messages are received class MultipleInstances generates message D.

**Pattern 14 : Multiple Instances (with a priori run time knowledge)**

This pattern is similar to pattern 13, however in this case, the decision about how many sub-processes are involved is not known until run-time. The logic about how many sub-processes to create is embedded in the code but is not executed until run-time.
Appendix 1: Workflow patterns

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class ObjectMaker extends Message

... int[] arrayOfIDs
...

method makeObjects
    arrayOfIDs = []
    // write one or more Active objects
    if (decisionA)
        aoA = new ActiveObjectA
        write(new aoA)
        arrayOfIDs.add(aoA.getID())
    if (decisionB)
        aoB = new ActiveObjectB
        write(new aoB)
        arrayOfIDs.add(aoB.getID())
    if (decisionC)
        aoC = new ActiveObjectC
        write(new aoC)
        arrayOfIDs.add(aoC.getID())
    // return an array of identifiers
    return arrayOfIDs // zero to 3 ids
```

Class ObjectMaker is a message containing executable code that is generated by an upstream task and then received by MultipleInstances. When the makeObjects method is invoked by MultipleInstances, ObjectMaker generates a number of sub-processes (Active Objects) depending on run-time decisions. The identity of these subprocesses is passed back to MultipleInstances which waits for messages to arrive from these sub-processes. When each sub-process terminates it generates a corresponding message for MultipleInstances. When all messages are received by MultipleInstances, message D is generated.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class MultipleInstances extends ActiveObject
```
Register(ObjectMaker)
...
method notify(message)

    if (message==ObjectMaker)
        messageArray = object.makeObjects()
        return

    // find, mark and count each task
    objectCount = 0
    for each of messageArray
        if (object==messageArray[i].id)
            objectArray[i].flag = true
            completedTaskCount++

    // all tasks received
    if objectCount == length(objectArray)
        write(new messageD)
        terminate

class ActiveObjectA extends ActiveObject
    Wait for a condition
    Write(new messageA)

class ActiveObjectB extends ActiveObject
    Wait for a condition
    Write(new messageB)

class ActiveObjectC extends ActiveObject
    Wait for a condition
    Write(new messageC)

**Pattern 15 : Multiple Instances (without a priori run time knowledge)**

This pattern is similar to patterns 13 and 14 but the knowledge about how many sub-processes to create is provided from outside of the code (eg. by a user).
Appendix I: Workflow patterns

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class ObjectMaker extends ActiveObject
    ...
    while (continue)
        waitForUserAction()
        write(new UserActivatedObject(++Counter))

Class MultipleInstances extends ActiveObject

    Register( ObjectMaker,
               UserActivatedObject,
               UserObject)
    ...
    method notify(message)

        if (message==ObjectMaker)
            objectMaker.active = true
            write(new objectMaker)

        if (message==UserActivatedObject)
            object.active = true
            write(new object)
            UserObjectCounter++

        if (message==UserObject)
            UserObjectCounter++

        // write the output object
        if (ExpectedObjectCount==UserObjectCounter)
            write(new messageD())

class UserActivatedObject extends ActiveObject
    WaitForCondition()
    Write(new UserObject())
```

Class MultipleInstances receives ObjectMaker (an Active Object) from an upstream task. MultipleInstances then activates ObjectMaker and generates to the space where
it is removed and executed. ObjectMaker generates a number of UserActivatedObjects in response to user input. When each UserActivatedObject is activated by a user it generates a UserObject for MultipleInstances. When MultipleInstances receives all expected UserObjects it generates message D for a downstream task.

**Pattern 16: Deferred choice**

The user decides which path a message will take.

Typical pseudo-code to implement this pattern in a GC environment is:

```java
Class DeferredChoice extends ActiveObject

    Register(messageA)

    method notify(message)

        if (userChoice() == B)
            write(new messageB())
        else
            write(new messageC())
        terminate
```

Upon receipt of message A DeferredChoice presents a set of choices to the user. In response to the users choices DeferredChoice generates a specific message.

**Pattern 17: Interleaved parallel routing**

A set of processes is generated in any order but not concurrently. When these processes are complete the results are collected and when all results are received processing continues.

Typical pseudo-code to implement this pattern in a GC environment is:
Appendix 1: Workflow patterns

Class StartInterleaving extends ActiveObject

Register(messageA)
...
int count = 0
...
Method notify(message)

    If count == 3
        Write(new messageE)
        // complete
    Else If (message==messageA)
        GenerateNextMessage()
    Else if (message==messageB2)
        GenerateNextMessage()
    Else if (message==messageC2)
        GenerateNextMessage()
    Else if (message==messageD2)
        GenerateNextMessage()
Method GenerateNextMessage()
    Count++
    Switch(getSomeArbitraryOption())
        Case 1
            Write(new messageB)
        Case 2
            Write(new messageC)
        Case 3
            Write(new messageD)

Class EndInterleaving extends ActiveObject

Register(messageB1,messageC1,messageD1)
...
int count = 0
...
Method notify(message)
    count++
    if (count == 3) // complete
write(messageE)

Class TaskB extends ActiveObject

    Register(messageB)
    ...
    method notify(message)
        Do some process
        Write(new messageB1)

Class TaskC extends ActiveObject

    Register(messageC)
    ...
    method notify(message)
        Do some process
        Write(new messageC1)

Class TaskD extends ActiveObject

    Register(messageD)
    ...
    method notify(message)
        Do some process
        Write(new messageD1)

Because StartInterleaving must know when EndInterleaving has received a message there are two ways this pattern could be implemented. Start and End can be implemented as a single class which means that communication between the two is internal. On the other hand, Start and End can be implemented as two classes and communication between the two can be carried out using GC.

Pattern 18 : Milestone

The milestone pattern outlines a way of testing whether a particular point in a workflow has been reached.

Typical pseudo-code to implement this pattern in a GC environment is:
Appendix 1: Workflow patterns

Class Milestone extends ActiveObject

    Register(messageA, messageTest)
    ...
    method notify(message)
        if (message == messageA)
            milestoneReached = true
        else if (message == messageTest)
            write(new messageResult(milestoneReached))

Class MilestoneTest extends ActiveObject

    Register(messageResult)
    ...
    While (true)
        WaitForUserAction()
        Write(messageTest)

        method notify(message)
            if (message == messageResult)
                complete

When message A is received by class Milestone a milestone has been reached. Class MilestoneTest tests this condition by generating a message for class Milestone to see if a message has been received. When ever MessageTest is received by Milestone, MessageResult is generated back to MilestoneTest.

Patterns 19 and 20: Cancel Pattern

The purpose of cancellation is to allow all or part of a workflow to be cancelled. This can be achieved using GC by ensuring that all tasks that are implemented are notified when a cancellation message arrives in the space. The cancellation message can be a general cancellation which cancels all tasks, a specific cancellation which cancels a
single task or a cancellation which cancels a category of tasks. Cancel is similar to the “poison pill” concept [16].

Typical pseudo-code to implement this pattern in a GC environment is:

```java
class SomeTask extends ActiveObject

    Register(cancelThisProcessMessage,
            cancelAllProcessesMessages)

    ...

    method notify(message)
        if ( object = cancelThisProcessMessage or
            object = cancelAllProcessesMessage)
            killThread
```

The class someTask registers for notification of two cancel messages. While both will have the effect of terminating the task, other tasks in the workflow will be notified of cancelAllProcessesMessage and will also terminate. This type of notification provides the opportunity to selectively cancel parts or all of the workflow.
9 Appendix 2: Taxonomy of services

The diagram in Figure 56 shows the taxonomy of services used in this thesis.

Figure 56 Taxonomy of services in this thesis
10 Bibliography


58. Wynn, M., et al., *Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets*. 2005, Queensland University of Technology: Brisbane, Australia.
