Journal of Grid Computing **1:** 199–217, 2003.
© 2004 *Kluwer Academic Publishers. Printed in the Netherlands.*

Triana Applications within Grid Computing and Peer to Peer Environments

Ian Taylor[1], Matthew Shields[2], Ian Wang[2] and Omer Rana[3]
[1]*School of Computer Science, Cardiff University, UK*
E-mail: i.j.taylor@cs.cardiff.ac.uk
[2]*Schools of Physics and Astronomy and Computer Science, Cardiff University, UK*
[3]*School of Computer Science, Cardiff University, UK*

Key words: GAT, Grid, problem solving environment, Web services

Abstract

An overview of the Triana Problem Solving Environment is provided – with a particular focus on the GAP application-level interface, for integration with Grid Computing and Peer-to-Peer infrastructure. GAP is a Java-based subset of the Grid Application Toolkit interface (being implemented in the GridLab project), and an outline of its current functionality, usage and mappings to three supported underlying middleware derivatives: JXTA, Web Services, and P2PS (a simplified Peer-to-Peer platform) are provided. The motivation behind the development of P2PS is given – emphasising its minimal, but effective Peer-to-Peer mechanisms that allow scalable, decentralized discovery and communication amongst cooperating P2PS peers within highly unstable environments. A summary of three application use cases illustrating the range of scenarios that such a system addresses is also provided.

Abbreviations: OGSA – Open Grid Services Architecture; P2P – Peer to Peer; NAT – Network Address Translator; API – Application Programmers Interface; GAT – Grid Application Toolkit; TCS – Triana Controlling Service; BPEL4WS – Business Process Execution Language for Web Services; WSFL – Web Sevices Flow Language; GAP – Grid Application Prototype; WSDL – Web Services Description Language; UDDI – Universal Description, Discovery and Integration protocol; UDDI4J – UDDI for Java; WSIF – Web Services Invocation Framework

1. Introduction

There has recently been an overwhelming interest in the field of Grid computing and the introduction of the Open Grid Services Architecture (OGSA) [8] has gained a significant input from both commercial and non-commercial organizations ([17] and [31]). Further, the standardization of the OGSI specification [24] has led to a convergence in the infrastructure adopted for Grid computing implementations. In contrast, Peer-to-Peer (P2P) technology has also gained popularity through services like Gnutella [13] and SETI@home [26]. Further, since the emergence of P2P infrastructures, such as JXTA [25] the number of organizations interested in P2P has increased significantly.

Although the underlying philosophies of Grid computing and P2P are different, for certain scientific applications they can be both considered enabling technologies for accessing the vast computing resources that are available through the Internet and mobile devices. Both approaches attempt to solve the same problem [9], that is, to create overlay structures for the underlying organizational structure of the Internet, but they differ in that they deal with users with contrasting computational and resource requirements. For example, whilst Grid computing connects relatively small numbers of *virtual organizations* [7] that can cooperate in a collaborative fashion, P2P applications can connect hundreds of thousands individual users that exist in highly unstable environments consisting of devices living at the *edges of the Internet* (highly transient and behind NAT, firewalls, etc.).

An overview of the Triana problem solving environment is first provided in this paper, followed

by details of particular distribution mechanisms that it uses for component execution. Section 2 provides the overview of Triana, and compares it with existing work on Grid Computing Environments. Section 3 outlines the various mechanisms that can be used for distributed computing within Triana – and includes descriptions of the GAP and GAT interfaces (part of the GridLab project [15]) that abstract Triana from the underlying middleware. Subsequently JXTA, P2PS and Web Service bindings for the GAP API are discussed. This section also describes how Triana can be used to support task farming. The key motivation here is to illustrate how Triana can make use of different distribution mechanisms, based on a common API. The GAP API is then examined in more detail in Section 4 with a simple example program, and the implementation of the various bindings, JXTA, P2PS and Web Service, illustrated using code samples for a small subset of the functions. Usage scenarios are subsequently presented in Section 5 – with a particular focus on how computational visualisation and steering, data management, and service discovery may be undertaken with Triana. Although these scenarios are primarily focused on astrophysics applications – the use of Triana is by no means restricted to this domain.

2. Related Work and Triana Overview

A Problem Solving Environment (PSE) is a complete, integrated computing environment for composing, compiling, and running applications in a specific area [11]. In many ways, a PSE is seen as a mechanism to integrate different software construction and management tools, and application specific libraries, within a particular problem domain. One can therefore have a PSE for financial markets [4], for gas turbine engines [6], etc. Focus on implementing PSEs is based on the observation that previously scientists using computational methods wrote and managed all of their own computer programs – however now computational scientists must use libraries and packages from a variety of sources, and those packages might be written in many different computer languages. Engineers and scientists now have a wide choice of computational modules and systems available, enough so that navigating this large design space has become its own challenge. A survey of 28 different PSEs by Fox, Gannon and Thomas (as part of the Grid Computing Environments WG) can be found in [10], and practical considerations in implementing PSEs can be found in Li et al. [19]. Both of these indicate that such environments generally provide “some backend computational resources, and convenient access to their capabilities”. Furthermore, workflow features significantly in both of these descriptions. In many cases, access to data resources is also provided in a similar way to computational ones. Often PSE and Grid Computing Environment is used interchangeably – as PSE research predates the existance of Grid infrastructure.

Based on the surveys above, Triana may be classified as a graphical Grid Computing Environment (PSE) (both a problem solving and a programming environment) – and provides a user portal to enable the composition of scientific applications. Users compose applications by dragging programming components (called units or tools) from toolboxes, and drop them onto a scratch pad (or workspace). Connectivity between the units is achieved by drawing cables – which also account for particular data types – that connect different units together. An overview of how Triana operates can be found in [27].

Although Triana shares some common features with existing Grid portal technologies, providing both a composition environment and a mechanism for the distribution of components, is a novel feature. Existing efforts often focus on one or the other of these aspects. Additionally, the composition environment is developed so that it can be used individually, if desired. An XML-based task graph is generated from the composition tool, and supports bindings for distributing components using Web Services or P2P technologies. Triana also requires the existence of a Triana execution environment to exist on each node that is to host a Triana service. This is also a significant difference from existing portal technologies; existing systems assume the presence of a hosting environment on resources.

Li and Baker [19] provide an extensive review of various Grid portals currently available. Based on their definition, a Grid portal provides “end users with a customized view of software and hardware resources specific to their particular problem domains”. In some ways, this definition shares common themes with that of a Grid Computing Environment. The focus in their work is primarily on Web-based portals – which therefore differ from the focus in Triana (where the focus is on catering for different distribution mechanisms, rather than just a single one). However, they emphasise three generations of portal technologies: generation 1 being focused on a graphical interface and the use of the Globus toolkit – and are primarily tightly coupled with Globus-based Grid middleware tools.

Generation 2 portals are aimed at specifying "portlets" – essentially user customisable services which can run on top of a web server. Grid Portlets are intended to be independent components that can utilise a number of different Grid middleware toolkits. This is the current state of affairs in portals, with GridSphere [23] being a commonly used toolkit to support the construction of such Portlets. Generation 3 involves the extension of the Portlet idea with semantic annotations. Currently Triana supports a concept similar to Portlets, in that each individual component present within the Triana toolbox has an XML-based interface. Such an interface can dynamically bind to a Web Services-based or a P2P-based distributed mechanism. Furthermore, the P2PS mechanism employed in Triana can be used to advertise components in the toolbox based on their interfaces. The XML description can also be extended with semantic properties if required.

Triana therefore provides the following novel features:

- Support for both a service (component) composition, and a service (component) distribution environment.
- Support for a distribution API that can be mapped to a number of different underlying middleware implementations. Three such bindings are discussed in Section 3, and include JXTA, P2PS and Web Services.
- Support for interoperability between Triana components described as Web Services, and standard units contained in the Triana toolbox. This is achieved via a common XML interface for these components.
- Definition of components based on an XML data model that can also encode semantic/non-functional properties associated with components (such as cost of access, ownership details, etc).
- Support for hierarchy – whereby components can be aggregated into groups, and either published in the toolbox, or distributed to a remote machine for execution.
- A toolbox of components that are pre-provided for use, when Triana is downloaded. These include signal processing, image processing, maths and audio functions.

Based on the survey provided in [10], no other Grid Computing Environment supports all of these features.

2.1. *Triana Overview*

Triana was initially developed by scientists in GEO 600 [12] to help in the flexible analysis of data sets, and therefore contains many of the core data analysis tools needed for one-dimensional data analysis, along with many other toolboxes that contain components or units for areas such as image processing and text processing. There are around 500 units with Triana covering a broad range of applications. Further, a recent development in Triana is the ability to dynamically discover and choreograph distributed resources, such as Web Services, to greater extend its range of functionality. Consequently, Triana can be used by applications and end-users alike in a number of different ways [28]. For example, it can be used as: a graphical workflow composition system for Grid applications; a data analysis environment for image, signal or text processing; as an application designer tool, creating stand-alone applications from a composition of components/units; and through the use of its "pluggable workflow representation architecture", allowing 3rd party tool and workflow representation to be easily incorporated into the architecture, for example the ability to parse WSDL and BPEL4WS.

The Triana user interface consists of a collection of toolboxes containing the current set of Triana components and a work surface where users graphically choreograph the required behaviour, rather than specifying it using source code. The modules are late bound to the services that they represent to create a highly dynamic programming environment. Triana has many of the key programming constructs such as looping (do, while, repeat until, etc.) and logic (if, then, etc.) units that can be used to graphically control the dataflow, just as a programmer would control the flow within a conventional program using specific instructions. Programming units (i.e., tools) include information about which data-type objects they can receive and which ones they output, and Triana uses this information to perform design-time type checking on requested connections to ensure data compatibility between components; this serves the same purpose as the compilation of a program for compatibility of function calls.

Triana has a modularized architecture [27] that consists of a cooperating collection of interacting components. Briefly, the thin-client Triana GUI connects to a Triana engine (Triana Controlling Service, TCS) either locally or via the network. Under a typical usage scenario, clients may log into a TCS, remotely compose and run a Triana application and then visualize the result locally – even though the visualization unit itself is run remotely. Alternatively, clients may log off during an application run and periodically log

back on to check the status of the application. In this mode, the Triana TCS and GUI act as a portal for running an application, whether distributed or in single execution mode.

Users are free to use the Triana components in a number of different ways. For example, the *conventional* operational usage is to graphically compose applications from collections of interacting units; examples of this usage are given in Section 5. However, other usages include using the generalized writing and reading interfaces for integrating third party components and workflow representations within the graphical interface, the mechanism that is used by our the Data Translation example given in Section 5 to import and compose the relevant Web Services. In this case, we have written a WSDL reader/writer for importing Web Services and BPEL4WS [3] reader/writers that can be used to import/export the components as a workflow. A workflow composed graphically in this fashion could be directly imported into another execution or scheduling environment without modification.

3. Distributing Triana Taskgraphs

The mechanisms that enable the distribution of Triana components are described. These include the Grid Application Toolkit (GAT), the Grid Application Prototype (GAP) interface, JXTA/P2PS and Web Services. The GAT interface is outlined in Section 3.1. A reduced version of this – the GAP interface – is then described in detail in Section 3.2, along with the motivation for why such an interface is needed. Based on these two distribution interfaces, Section 3.3 discusses how the actual service distribution takes place, and the types of distributed computing techniques that can be supported (parallel, pipeline, etc). The use of control units is then discussed in Section 3.5 to demonstrate how these may be used to manage component execution (such as support for loops, check for conditionals, etc.).

3.1. *The GAT*

The GAT interface provides a generalised collection of calls to shield Grid applications from implementation details of the underlying Grid middleware, and is being developed in the European GridLab project [1, 2]. The GAT utilises *adaptors* that provide the specific bindings from the GAT interface to the underlying mechanisms that implement this functionality. For example, a *move_file* command may have many GAT adaptors that implement this functionality depending upon the particular execution environment used, such as GridFTP, JXTA pipes or a local *cp* command. GAT may be referred to as *upperware*, which distinguishes it from middleware (which provides the actual implementation of the underlying functionality). Until recently, application developers typically interact with the middleware directly. However, it is becoming increasingly apparent that this transition from one type of middleware to another is not a trivial one. Using interfaces like GAT, migrating from one middleware environment to another is easier, and typically achieved by setting an environment variable. This is illustrated in the next section where we have implemented an adaptor to bind to P2P middleware for operating in P2P environments as well as the Grid environments supported directly by GridLab. This means that exactly the same Triana implementation can be used within both environments transparently.

3.2. *The GAP Interface*

The Grid Application Prototype Interface (GAP Interface) is a generic application interface providing a subset of the GAT functionality. It is middleware independent, with bindings provided for different Grid middleware such as JXTA and Web Services, as illustrated in Figure 1. This model means that it should be possible to seamlessly switch applications (such as Triana) using the GAP to work with new Grid middleware, such as OGSA. One key focus for the GAP is to support Triana P2P interactions on the "Consumer Grid" [29] for running massively parallel high-throughput codes.

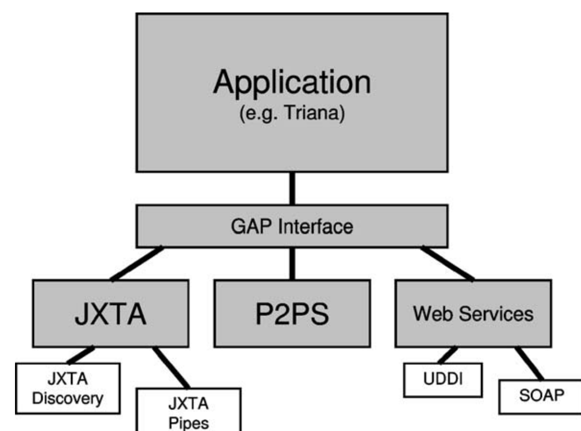

Figure 1. The GAP Interface provides a middleware independent interface for developing Grid applications.

Part of the motivation behind the GAP Interface is as a stopgap to enable us to develop distribution mechanisms within Triana while the GridLab GAT is being developed. When the GridLab GAT becomes available the GAT-API will replace the GAP Interface within Triana and should enable Triana to make use of the advanced security, logging and other GridLab services. However, the GAP Interface will live on, both as a simple interface for prototyping Grid and P2P applications, and as an adaptor within the GridLab GAT architecture providing various discovery and communication capabilities. Currently there are three GAP bindings implemented:

JXTA – The original GAP Interface binding was to JXTA [25]. JXTA is a set of protocols for Peer-to-Peer discovery and communication originally developed by Sun Microsystems. Although we achieved some initial success with JXTA, we have since had problems with the speed and reliability of our JXTA binding. See Section 4.2.

P2PS – a lightweight Peer-to-Peer middleware. See Section 4.3.

Web Services – The most recent GAP binding allows applications to discover and interact with Web Services – using the UDDI registry [32] and the Web Service Invocation Framework (WSIF) [34]. See Section 4.4 for details.

3.3. *Triana Service Distribution*

Computational Grids typically include a number of heterogeneous resources owned and managed by different administrators. Each computing resource may offer one or more services and each service could be a single application or a collection of applications. The service paradigm has become an important abstraction in Grid computing, primarily since the release of OGSA.

Triana uses this same level of abstraction for representing its core services. However, within Triana, such services may be represented in many different ways, e.g., Web Services, P2PS or JXTA services, and since the mechanism that provides the interaction is supplied by the GAP interface, this gives us the ability to communicate with these heterogenous set of Triana services in the same way. Triana services, regardless of their implementation technology, must support the following criteria:

Service Parsing – provides interfaces for reading/writing service descriptions and (optionally) for representing a collection of these within a workflow.

Discovery/Communication – provides support for discovering and communicating with services. For example, the GAP interacts with UDDI and uses SOAP/WSIF for this mechanism, when using a Web Services implementation.

A user logged into a Triana Controlling Service (TCS) can elect to distribute parts of their workflow to be executed on remote machines by distributing the code to two types of Triana services:

Generic – a service that can execute any workflow passed to it on Grid resources. A Java Sandbox may provide limited security for applications on remote resources.

Specific – a particular Triana network (workflow) may be constructed and then deployed as a service. Such a service can only thereafter run this particular set of units. Using this method, all Triana units may be deployed as Web Services.

As well as distributing workflow to single remote machines, a user can elect to use a custom distribution policy for sending sections of workflow to more than one machine (in parallel for example). Custom distribution within Triana is makes use of compound Triana units, which we call Group Units. Group units are aggregates that contain a number of interconnected units and have the same properties as conventional ones. They have input/output nodes, parameters etc., and therefore can be connected to other Triana units using the standard mechanism. In order to distribute a Group unit, a user must specify the custom distribution policy for that group. There are currently two distribution policies:

Parallel is a "farming out" data parallel mechanism (see Figure 2) and generally involves no communication between hosts.

Pipeline involves distributing the group vertically, i.e., each unit in the group is distributed onto a separate resource and data is passed between them in a pipelined fashion.

Groups can be hierarchic, and each group can have its own distribution policy – allowing for complex hierarchical distribution mechanisms, as illustrated in Figure 2. In this figure, one Triana Service distributes its group to three other Triana services using the parallel distribution policy; then each of these Triana services act as a gateway and distributes their task-graph (implemented by a subgroup) to two other services using the pipeline distribution policy.

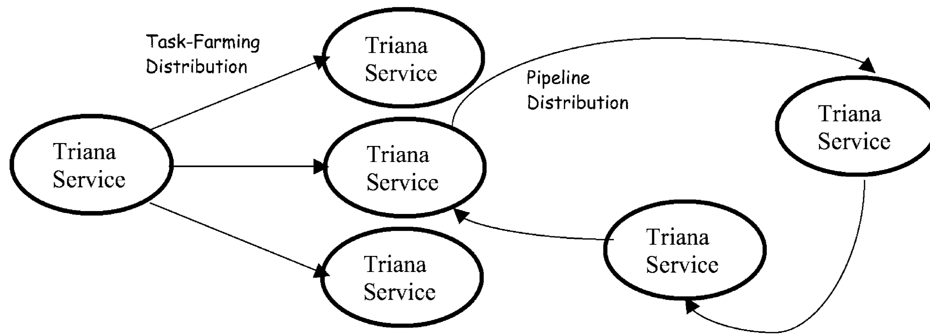

Figure 2. Example Triana distribution: a service distributes a task-graph to three other Triana services using task-farming then each of these distributes their task graphs to another two services using a pipelined approach.

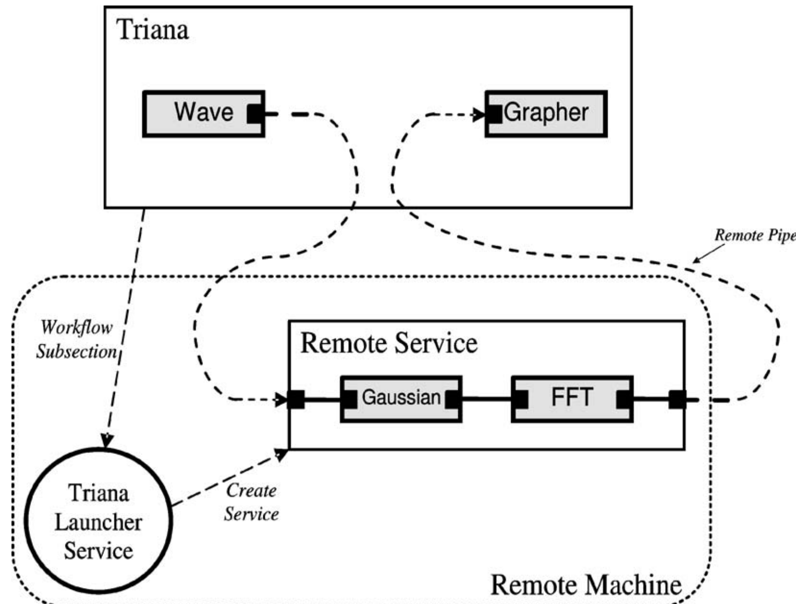

Figure 3. Illustrates the flow of data from the controlling Triana service and the remote Triana services that implement a particular binding of the GAP interface.

3.4. *Triana Distribution Using the GAP Interface*

If a workflow to be distributed in Triana consists of more than a single task, then a Group task must be created prior to distribution. A user subsequently selects the resource on which this Group task must run, and the GAP Interface is used to automatically launch each workflow subsection as a remote service. A Triana Launcher Service is required on a remote machine for executing the workflow, and is a GAP-based service that, when sent a workflow subsection (serialized in XML) via its control pipe, uses the GAP Interface to launch that subsection as a new Triana service. The actual form the Triana Launcher Service and the new Triana services take depends on the GAP binding employed (see Section 4); for example, if the Web Services/GAP binding is used then both the Triana Launcher Service and the new Triana services are created as Web Services. Once remote services have been launched, they replace the equivalent local tasks in the users' workflow. Data is transferred to/from the new remote services via their input and output pipes, as illustrated in Figure 3.

A key feature is the capability to advertise workflow subsections (if authorized), launched as remote Triana services, using the GAP bindings (e.g., P2PS advertisement for the P2PS/GAP binding or UDDI for the Web Services/GAP binding (see Section 4)). Once advertised these Triana services can be discovered and invoked by both other instances of Triana and by non-Triana related applications. This is a powerful feature

as Triana can be used to quickly create and launch a wide range of composed Triana algorithms as services.

When Triana is started it automatically uses the GAP Interface to search for existing published Triana services, and any that are found are inserted into the users tool tree alongside the existing local tools. Additionally, a "Discover Services" option may be used to issue a GAP discover services call. When a remote service is located it is inserted into the users tool tree. Once discovered, remote tools can be dragged and connected into the users workflow in exactly the same manner as local tools. However, when an input/output cable is connected to a remote task, Triana connects an input/output GAP pipe to the remote service instead of a local cable.

3.5. *Control Tasks*

As outlined in the previous section, Triana also provides a standard mechanism for distributing group tasks across multiple machines either in parallel, as a pipeline, or using some other custom distribution topology. In Triana each group task is accompanied by a control task that receives the data input to the group before it is passed to the tasks within the group, and also receives the data output from the group before it is sent on from the group. The original use of control tasks was to provide looping over the group; however this has been extended to allow control tasks to specify and control the custom distribution of groups. As control tasks are standard Triana units, Triana users can implement custom distribution policies to meet their distribution requirements.

A control task in a custom group distribution is used to specify the distribution topology. This is done by splitting and cloning the original group of tasks into a number of sub-workflows, each of which is launched by Triana as a remote service using the GAP Interface. The machines on which these remote services are launched can either be automatically chosen by Triana or explicitly selected by the user. As well as specifying the distribution groups, the control task also specifies the connections between the remote services and between the local control task and the remote services. Triana uses this connection specification to create the input/output pipes that link the local workflow and the remote services into a fully connected distributed workflow. In Figures 4 and 5 we contrast a group of tasks distributed in parallel to two remote machines with the same group distributed as a pipeline.

Secondly, a control task in a group distribution controls the distribution of data to the remote services. In group distribution, the control tasks can determine which remote service processes a data item it receives

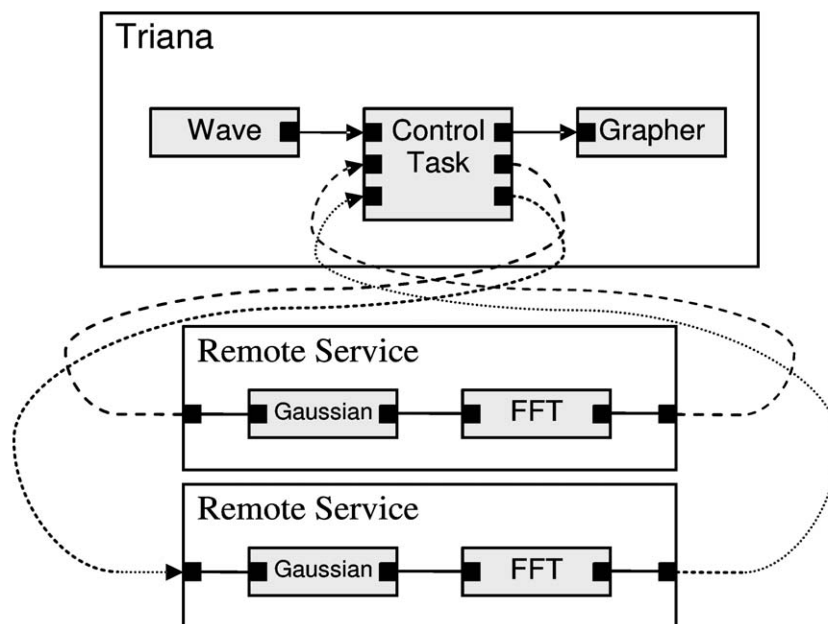

Figure 4. Triana dynamically rewires the taskgraph at run time to connect to the remote peers it has discovered through the GAP interface discovery calls. This figure shows the reconnection to two independently running services that can be used in for high throughput, or parallel computation.

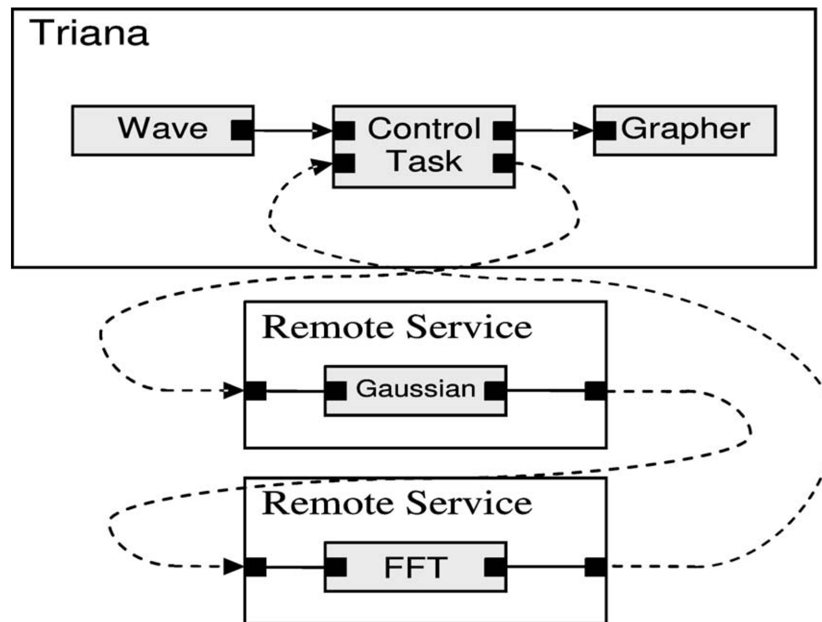

Figure 5. Triana can also support the dynamic pipelined connectivity of a collection of cooperating remote peers.

through its choice of output node – as illustrated in Figure 4, i.e., the control task is acting as a simple scheduler, distributing the data over a number of parallel remote processes.

3.6. *Operational Overview: Using Triana to Task Farm*

Once a particular workflow has been created, sections of this can be grouped and then distributed to running Triana services using any of the distribution policies described in Section 3.3. For the Galaxy formation example (Section 5), the parallel distribution policy is used to task farm the data sets across all available nodes, in order to speed up the recalculation of the visualization. To achieve this, first the taskgraph that needs to be distributed is selected, then the data buffering tool and the processing unit (for recalculating the new viewing angle) are distributed. A group unit is then constructed from this combination representing the code that is to be duplicated on to the available resources. Right clicking on the group then brings up a menu where a user can enable the distribution of the group unit. Once this is selected, the window in the left side of Figure 6 appears prompting the user to enter the distribution policy. HTCParallel: the High Throughput Computing Parallel distribution implementation within Triana is selected in this instance. The window shown at the right side of Figure 6 is subsequently displayed, and shows the available services along with a number of other options. For example, users can select *Auto Distribution* allowing Triana to automatically utilize the available services, or *Custom Distribution* to allow a specific subset of the servers listed to be chosen. Also, a *Sequence Policy* can be selected that specifies how Triana will re-assemble the data items when they are returned to the client. The order of the returned data is not guaranteed by the GAP, so the sequence policy allows Triana to associate an ordering on the data before it is distributed and then use that order to reassemble the data after processing.

3.7. *Summary*

This section of the paper has briefly covered how Triana provides the ability to perform distributed computing tasks in a middleware independent manner. We introduced the GridLab GAT and our functional subset, the GAP interface, used by Triana to abstract different middleware implementations. We briefly discussed some of the implementations of the GAP bindings and functionality, including communication, service discovery and service publishing, these are extended in the next section. Finally we introduced the ideas of Control Tasks and distribution policies which allow users of Triana to perform distributed and data parallel processing.

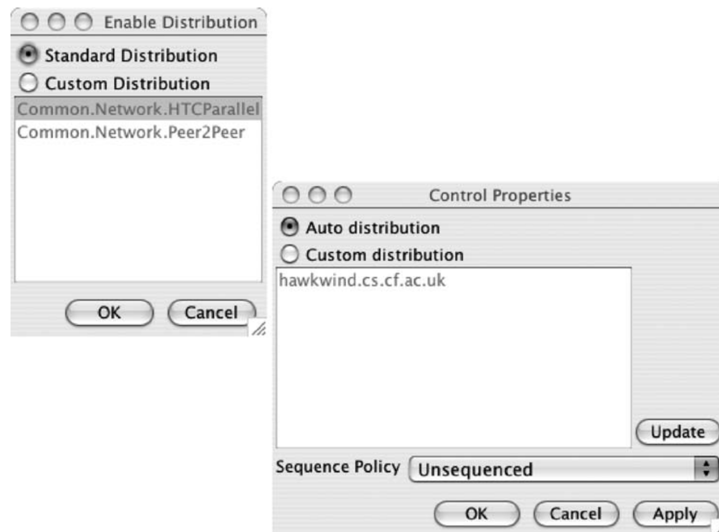

Figure 6. Distribution mechanism selection window.

4. The GAP Bindings

This section describes how we map from the GAP interface definitions to the various underlying bindings that are currently supported. The GAP interface contains a number of calls that are primarily focused on advertising, discovering and communicating with remote services. It was inspired by the GAT specification and has received input from a number of different groups. It can support underlying client/server, brokered or decentralized environments through its generalized set of application-level calls, specifically chosen to provide the right level of abstraction away from their implementation. The four subsections within Section 4.1 provide an overview of the calls contained within this interface including a simple usage example. The following three Sections, 4.2 to 4.4, then describe the particular bindings in more detail, including code snippets of how we translate the abstract GAP calls into JXTA pipes, SOAP invocations or P2PS discovery calls.

4.1. *The GAP Interface Illustrated*

The GAP Interface is based on a series of Java *interface* classes with concrete implementations that form the GAP bindings. The core interface is the `Peer` interface that contains the main group of functions for the GAP. The `Peer` interface can be split into four functional areas:

- Service Creation and Discovery
- Pipe Creation and Discovery
- Message Communication
- Information

4.1.1. *Service Creation and Discovery*

This group of functions is responsible for the creation of new services, the advertisement of those service instances so they can be found, and the discovery mechanism for finding them. The function calls contained in this group are

```
public Peer createService(String name)
    throws PeerException;

public void advertiseService() throws PeerException;

public void addServiceDiscoveryListener(String name,
    ServiceDiscoveryListener listener);

public void removeServiceDiscoveryListener(String
    name, ServiceDiscoveryListener listener);

public void locateServices(String name);

public RemoteServiceInfo[] getServices(String name)
    throws PeerException;
```

For example to create an instance of a service we would use the function `createService` to create a service with a given name. To advertise our newly created service we have the function call, `advertiseService`. Service discovery is performed *asynchronously*, through Java *listeners*,[1] passing the

[1] A Java Listener is an implementation of the publish-subscribe or *observer* design pattern. An *observer* registers with a *subject* and subsequently receives *notifications* about *state* changes. In the GAP it is used to implement non-blocking service discovery calls.

name of the service we wish to discover as a parameter of the `locateServices` function. To make use of all the services discovered at this point, we can either use `getServices` to return a list, or subscribe to service discovered events, `addServiceDiscovery Listener`.

4.1.2. *Pipe Creation and Discovery*

The next group of functions are concerned with the creation and discovery of communication channels or "pipes" between a pair of peers. The function calls in this group are

```
public InputPipe createControlPipe(String name,
    MessageListener listener) throws PeerException;

public InputPipe createInputPipe(String name,
    MessageListener listener) throws PeerException;

public void advertiseInputPipe(InputPipe inpipe)
    throws PeerException;

public void addPipeDiscoveryListener(String name,
    PipeDiscoveryListener listener);

public void removePipeDiscoveryListener(String name,
    PipeDiscoveryListener listener);

public void locatePipes(String name);

public RemotePipe[] getRemotePipes(String name)
    throws PeerException;

public InputPipe[] getInputPipes(String name);

public OutputPipe[] getOutputPipes(String name);

public OutputPipe connectOutputPipe(RemotePipe remote)
    throws PeerException;

public OutputPipe connectOutputPipe(String name,
    long timeout) throws PeerException;
```

The creation and advertisement of communication pipes is similar to that of the services that the pipes will connect. We have a constructor function that takes the pipe name and a `MessageListener` and an advertise function that publishes the constructed pipe. Although they are functionally the same we differentiate between *control* and *input* pipes. The former is used for sending instructions such as set up details to a peer and the latter is the data pipe.

As with the peers themselves the discovery mechanism is asynchronous and we can complete the connection between the local `InputPipe` and the remote `OutputPipe` either by using a reference to the `RemotePipe` or by name.

4.1.3. *Message Communication*

The message communication group of functions is used for sending and receiving messages between peers. The functions are

```
public String send(OutputPipe pipe, Object object)
    throws PeerException, IOException;

public void send(String pipename, Object object)
    throws PeerException, IOException;

public void addMessageListener(InputPipe pipe,
    MessageListener listener);

public void removeMessageListener(InputPipe pipe,
    MessageListener listener);
```

Using these functions is straight forward once the services or peers and the communication pipes have been created. We use the `send` function to send a Java object down a named `OutputPipe` or a referenced `OutputPipe` object. There is no corresponding receive function as all messages are asynchronous. To receive a message a peer registers itself to receive `MessageEvent` items using the `addMessageListener` function and then waits for the messages to appear on it's `InputPipe`.

4.1.4. *Information and Serialization*

The final group of functions are used to get information about a service or to serialise the information about a local service so that it can be sent along a control pipe and used to get a reference to the local `RemotePipe` remotely.

4.1.5. *Simple Example*

A simple example will illustrate the basic use of the GAP interface. The example listed below is a simple peer that performs the functionality of a "chat" service. Once started the service will attempt to discover other chat services and establish a communication pipe to them. Once the pipe has been established a simple user interface allows the user to chat to the other services.

The `Chat` class implements two of the GAP event listener interfaces, `ServiceDiscoveryListener` and `MessageListener` each of which has a single method that must be implemented. The class has three *class* variables

```
public static String CHAT_SERVICE = "ChatService";
private Peer peer;
private ChatWindow window;
```

The *static* string, `CHAT_SERVICE`, is the name that all instances of the chat service will use to identify themselves. The `Peer` variable is the GAP Peer that does all

the work and ChatWindow is a simple GUI object that is not included here for brevity.

The constructor initialises the GUI and creates a *concrete* instance of a Peer, in this case a P2PS instance. Next we create a ControlPipe, advertise the service and add the instance of the Chat object as a ServiceDiscoveryListener of the Peer.

```
// construct new service
peer = new P2PSPeer().createService(CHAT_SERVICE);

// create control pipe and advertise service
peer.createControlPipe(CHAT_SERVICE, this);
peer.advertiseService();

// listen for other chat discovery
peer.addServiceDiscoveryListener(CHAT_SERVICE, this);
```

When the discovery mechanism finds another chat service, the listener call back mechanism calls the implemented function from the ServiceDiscovery Listener interface.

```
// Called when a service is discovered
public void serviceDiscovered
  (ServiceDiscoveredEvent e) {
  try {
      // create output pipe to discovery control pipe
      OutputPipe outpipe = peer.connectOutputPipe(
          e.getServiceInfo().getControlPipe(
              CHAT_SERVICE));

      // send introductory message
      peer.send(outpipe,
          "#" + window.getChatName() + "is online");
  }
  catch(Exception except) {
      except.printStackTrace();
  }
}
```

Here, when a new service is discovered, we get its control pipe from the service's information context and connect to it. This enables a uni-directional connection to the remote service for passing chat messages though. We then simply notify the other participant that we are on-line.

The Chat object also implements the Message Received interface and when a message is received on the InputPipe the following implemented interface method is called to display the chat message on the user interface.

```
// Called when a message is received by the control
pipe.
public void messageReceived(MessageEvent event) {
    // print message to chat window
    window.print((String) event.getMessage().
                 getObject());
}
```

When the user types a message on the user interface and sends it the following method is called to send the message to all the other chat clients.

```
// Called by the chat window when the user sends a
message public void sendMessage(String text) {
    try {
        // send message to ALL pipes named
        CHAT_SERVICE
        peer.send(CHAT_SERVICE, text);
    }
    catch(Exception except) {
        except.printStackTrace();
    }
}
```

Note here that the GAP supports one-to-many communication channels since it could have just as easily discovered many CHAT_SERVICE peers. One call to send will propagate this message to all peers that advertise themselves as a chat service. In this way, simple peer grouping can be achieved.

If we wanted to produce this same example using a different implementation of the GAP interface, the only code we would have to change is the Peer constructor in the Chat constructor function of the code. The new code would be

```
peer = new JxtaPeer().createService(CHAT_SERVICE);
```

The following three sections will examine our implementations of the GAP interface by examining a subset of the Peer functions. Specifically the asynchronous locateServices function and the send message function.

4.2. *The JXTA Binding*

The JXTA binding for the GAP implementation is called *JxtaServe*. The concrete implementation of the GAP Peer interface within JxtaServe is a class called JxtaPeer. If we look at the asynchronous locate-Services function, the implementation in JxtaPeer, below, calls an equivalent function in the JXTA library called getRemoteAdvertisements.

```
public void locateServices(String name) {
    PeerGroup group;
    ServiceDiscoveryInfo info =
        new ServiceDiscoveryInfo(name);
    DiscoveryService dserv =
        group.getDiscoveryService();
    dserv.getRemoteAdvertisements(
        info.getPeerID(),
        DiscoveryService.ADV,
        "Name",
        info.getName(),
        DISCOVERY_THRESHOLD,
        listener);
}
```

The send function implementation also maps to an equivalent JXTA library function. All the work in the JxtaPeer function deals with creating an appropriate JXTA message container to wrap the Java Object data we wish to send. The object is serialised before being added as the JXTA message content.

```
public String send(OutputPipe pipe, Object object)
    throws PeerException {
    // Serialize the data object, code omitted
    byte[] serializedObject = byteout.toByteArray();

    // Create the Jxta message
    msg = new net.jxta.endpoint.Message();

    // Add data element
    msg.addMessageElement(
        new ByteArrayMessageElement(
            DataTag,
            new MimeMediaType("text/plain"),
            serializedObject, null));

    // Add Peer info
    msg.addMessageElement(... PeerIdTag, ...,
        peerinfo.getID());

    // Add ServiceInfo
    msg.addMessageElement(... ServiceNameTag, ...,
        serviceinfo.getName());

    // Add adverts for any ControlPipes
    PipeAdvertisement[] pipeads =
        serviceinfo.getControlPipeAdvertisements();
    for (int count = 0; count < pipeads.length;
        count++)
        msg.addMessageElement(... ServiceControlTag,
            ..., pipeads[count].getDocument());

    // Send the message
    ((JxtaOutputPipe) pipe).getOutputPipe().send(msg);
}
```

4.3. *The P2PS Binding*

Our first attempt at using P2P computing within Triana used the Java implementation of the JXTA protocols [25]. We found that although these protocols were well thought out, the implementation was very hard to use with a steep learning curve. In addition with the early versions of the implementation we were using, we had problems with reliably and repeatably discovering and communicating with peers and creating communication pipes. We realised that much of the functionality in the JXTA implementation was not necessary for our needs and so we developed P2PS as a lightweight alternative.

P2PS (P2P Simplified) is a lightweight P2P infrastructure based on XML advertisements and messaging. As the name suggests, P2PS aims to provide a simple infrastructure on which to develop P2P style applications, without the complexity of other similar architectures such as JXTA and JINI [18].

Like JXTA, the P2PS infrastructure employs XML in its discovery and communication protocols, and is independent of any implementation language or computing hardware. Assuming that suitable P2PS implementations exist, it should be possible to form a P2PS network that includes everything from super-computer peers to PDA peers. Furthermore, communication within P2PS is not tied to any single transport protocol, such as TCP, and can be extended to include new protocols, such as Bluetooth. The current reference implementation of P2PS is written in Java, and handles pipe communication over both TCP and UDP by default.

Although P2PS is not an implementation of the JXTA protocols, its architecture is inspired by that of JXTA. However, P2PS focuses only on the core elements required for peer discovery and pipe-based communication.

At the core of P2PS is the notion of a pipe, a virtual communication channel that is only bound to specific endpoints at connection time. When a peer publishes a pipe advertisement it only identifies the pipe by its name, id and the id of its host peer. A remote peer wishing to connect to a pipe must query an endpoint resolver for the host peer in order to determine an actual endpoint address that it can contact. In P2PS a peer can have multiple endpoint resolvers, with each resolving endpoints in different transport protocols or returning relay endpoints that bridge between protocols (e.g., to traverse a firewall).

Pipes advertised in P2PS are typed, with the standard types being unidirectional, bidirectional and discovery. Discovery pipes enable peers to broadcast advertisements to other peers; a typical implementation of a discovery pipe would be UDP multicast. Extensions to the standard pipe types are allowed in P2PS, with obvious extensions being reliable and secure pipes. In addition to named pipes, a peer can also advertise services; a service in P2PS simply being a named collection of pipes.

In terms of the GAP Interface, a binding between the GAP and P2PS is implemented that allows P2PS to be used via the GAP Interface. This means that it should be possible to seamlessly use applications developed on top of the GAP Interface in a P2PS network just by switching the GAP binding used (in the same way applications can be switched to a Web Service binding, a JXTA binding, etc.). In addition

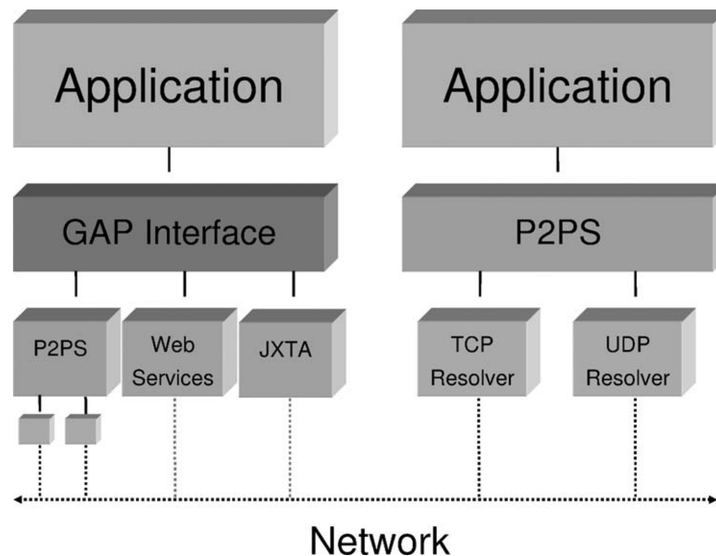

Figure 7. The architecture an application using P2PS via the GAP Interface compared with an application using P2PS directly.

to enabling an application to be switched between middleware bindings, using P2PS via the GAP Interface also shields application developers from handling P2PS advertisements directly; however, on the downside, much of the extensibility of P2PS is lost behind the GAP Interface. In Figure 7 we compare the architecture of an application using P2PS via the GAP Interface with an application using P2PS directly.

The concrete implementation of the GAP `Peer` interface within P2PS is a class called `P2PSPeer`. If we look at the asynchronous `locateServices` function, the implementation in `P2PSPeer`, below, uses a `DiscoveryService` in the P2PS library to search for services.

```
public void locateServices(String name) {
    ServiceDiscoveryInfo info =
        new ServiceDiscoveryInfo(name));
    ServiceQuery query = (ServiceQuery) peer.
        getAdvertisementFactory().newAdvertisement(
            ServiceQuery.SERVICE_QUERY_TYPE);
    query.setQueryPeerID(info.getPeerID());
    query.setQueryServiceName(info.getName());

    peer.getDiscoveryService().publish(query);
}
```

The `send` function implementation in `P2PSPeer` maps to a send function on a `P2PSOutputPipe`. Unlike the complicated message creation in the `JxtaPeer` implementation, the serialization, wrapping around the data object and message creation is handled by the `P2PSMessage` class.

```
public String send(OutputPipe pipe, Object object)
    throws PeerException, IOException {
    P2PSMessage message = new P2PSMessage(object,
        peerinfo, serviceinfo);
    ((P2PSOutputPipe) pipe).getOutputPipe().send(
        message.toByteArray());
}
```

4.4. *The Web Services Binding*

The Web Services implementation of the GAP interface, *WServe*, makes use of UDDI and a `UDDIProxy` object to locate services. The `Peer` interface implementation, `WSPeer` has the following `locateServices` method:

```
public void locateServices(String name) {
  ServiceDiscoveryInfo info =
      new ServiceDiscoveryInfo(name));
  int searchType =
      ((WSServiceDiscoveryInfo) info).getSearchType();
  String searchName =
      ((WSServiceDiscoveryInfo) info).getSearchName();
  Vector vectorServiceNames = new Vector();
  vectorServiceNames.addElement(new Name(searchName));

  UDDIProxy proxy = initUDDIProxy();
  ServiceList serviceList =
      proxy.find_service(null, vectorServiceNames,
      ...);

  // Code omitted
}
```

The `send` function implementation makes use of the *Axis* framework.[2]

[2] Apache Axis is an implementation of the SOAP "Simple Object Access Protocol" submission to W3C.

```
public String send(OutputPipe pipe, Object object)
    throws PeerException, IOException {
    String sendid = String.valueOf(sendcounter++);
    threads.addTask(new AXISInvoke(this,
        (WSRemotePort) pipe, object,
        isConvertObjectArrays(),
        isCustomSerialization(), sendid));

        return sendid;
}
```

In this section we have illustrated the GAP interface through the use of a simple "chat" client example that makes GAP function calls to discover other chat services and then send and receive messages between them. We outlined the major GAP function call groups and examined three different binding implementations by looking at two functions, `locateServices` and `send`. Although the binding implementation technologies are very different the GAP abstraction of service discovery and message communication can be applied to all three. Triana currently uses each of the bindings in isolation but the GAP should allow the inter-operation of any service using any binding, so that a P2PS service could talk to a Web Service or a JXTA service. It is easy to see how, with very little code changes, an application can use whichever binding it requires.

5. Triana Application Scenarios

Application scenarios are presented in this section to illustrate how Triana units and distribution mechanism can be used in practise. Although two of the three applications mentioned here are particularly focused on astrophysics, the data management and unit execution mechanisms may also be utilised in other application domains.

5.1. *Galaxy Formation Visualisation*

Galaxy and star formation using smoothed particle hydrodynamics generates large data files containing snapshots of an evolving system stored in 16 dimensions. Typically, a simplistic simulation would consist of around a million particles and may have a raw data frame sizes of 60 Mbytes, with an overall data set size of the order of 6 GBytes. The dimensions describe particle positions, velocities, and masses, type of particle, and a smoothed particle hydrodynamic radius of influence. After calculation, each snapshot is entirely independent of the others allowing distribution over the Grid for independent data processing and graphic generation (see [30]).

A user of the galaxy formation application would like to view the chronological changes in the galaxy as an animation and be able to alter their point of view, changing the two dimensional view of the three dimensional space. The application consists of three main data management activities, which are implemented as three individual units within Triana:

- **File Parsing** – data files are parsed according to their format and the data loaded into data structures, each data segment representing a distinct time frame within the animation
- **Data Set Projection** – the 3D data sets are projected down onto a 2D plane from a viewpoint, this calculation comprises the majority of the required processing. A Triana component is used to alter the position the user views the galaxy forming from, using a convenient three *scrollbar* user interface, representing the X, Y and Z coordinates. Setting these parameters once on the client simultaneously updates any Triana services that have been task-farmed to perform parallel processing on the data. The user selects the precise viewpoint and presses the start button to initiate the recalculation of the view of the galaxy.
- **Visualization** – the 2D frames are returned to the client for viewing

Triana helps in this example by allowing the user to distribute the *Data Set Projection* unit over all available Triana services and perform parallel processing on the discrete data frames without having to write a single line of parallel code.

5.2. *Inspiral Search Algorithm*

Einstein's theory of General Relativity predicts the existence of "gravitational waves". We have indirect evidence for the existence of gravitational waves but no direct observation has so far been made. Such waves are generated by compact binary stars orbiting each other – until their collision. Some Laser interferometric detectors such as GEO600, LIGO and VIRGO should be able to detect the waves from the last few minutes before collision. A gravitational wave passing through the interferometer causes displacements of the mirrors and a shift in the interference pattern. The amplitude of the displacement will be extremely small. To search for an inspiral signal, the detector output is examined for signals of particular shape. This shape

is called a template and it is constructed using theoretical knowledge about relativistic binary systems. It is determined by its family of parameters, the most important of which are the masses of the compact objects.

For the inspiral search application, the search algorithm works by correlating the data with the templates, a technique known as *template matching* or *matched filtering*. The correlation is achieved by using the *fast correlation* algorithm by taking the Fourier transform of both the template and the data, multiplying them together, then taking the inverse fourier transform of the result. There are typically tens of thousands of templates (in each bank), each containing different parameters defined at a certain granularity within the search space. The key factor is to be certain that the search space is fine-grained enough to catch the incoming waves. Existing C code is currently used to calculate the bank parameters and write them to a text file. This is read, the templates are generated and the correlations are performed. For a modest search, we would need to have a computing resource capable of speeds in the range of 10 Gigaflops to keep up in real time with an on-line search. For example, the gravitational wave signal is sampled at 16 kHz and sampled at 24-bit (stored in 4 bytes). However, the meaningful frequency range is up to 1 KHz and therefore a sampled representation of this contains 2,000 samples per second. The real-time data set is divided into chunks of 15 minutes in duration (i.e., 900 seconds) which results in a 7.2 MB of data ($4 \times 900 \times 2000$) being processed at a time. The algorithm has the following steps:

File Transferring – this data is transmitted to a node, currently this is achieved by a central coordinator, i.e., the client but we are currently in the process of converting this mechanism into a decentralized distributed process.

Processing – the node initialises, i.e., generates its templates (a trivial computational step) and then it performs fast correlation on the data set with each template in a library of around 10,000 templates. This process takes about 5 hours on a 2 GHz PC running a C program (around 20 dedicated PC's with fast communication abilities would need to be employed full-time to keep up with this dataset). To perform a search in real time we must filter each segment of data through the bank before the next segment comes in.

Results – Results containing a minimal amount of data, e.g., the GPS second and the correlation ratio of the detected binary, are returned to the client if detected. Such events, typically of the order of a few per year are a trivial but important step. On the client side, we will employ the use of various notification schemes upon successful detection, e.g., email notifiation, SMS notification and screen alerts, which are readily available within the system.

The Inspiral Search algorithm is an indication of the scale of problem that can be addressed by Triana. The implementation in Triana [5] uses approximately fifty separate algorithmic components connected together to form a work flow task-graph, and is shown in Figure 8. Here, we really see Triana used as a graphical programming tool that reuses much existing code in order to generate new methods for analysing data. The enormous advantage to the application scientist of this approach is that experimenting with new novel detection methods is trivial and typically only involves inserting or replacing a unit and slightly rewiring the existing task-graph. This can all be accomplished within the environment. The traditional steps of code, compile, execute are reduced to insert component and execute.

5.3. *Data Translation*

Simple string input tools are often useful in many application scenarios, often for translating one type of data source into another. We demonstrate one such, based on two Web Services: read_bible [35], a service for extracting specified verses from the bible; and BabelFish [35], a service that converts text between two languages (English to French in this example). The result from this workflow, specified verses of the bible converted into French, is displayed in the standard Triana string viewer component. Such an application illustrates how third party services (in this case obtained from `xmethods.net`) may be combined together using Triana.

Figure 9 illustrates the workflow in Triana. In this case, service descriptions are specified as a Web Service Description Language (WSDL) document, and can be published into a UDDI registry [32]. When a WSDL description of a Web Service is discovered or imported, it is parsed by Triana and a Triana tool representing each of the operations available on the service is generated. The input and output nodes on each tool represent the input message parts required by the Web Service and the message parts returned from the Web Service operation respectively; for example, if a Web Service operation requires two input message parts

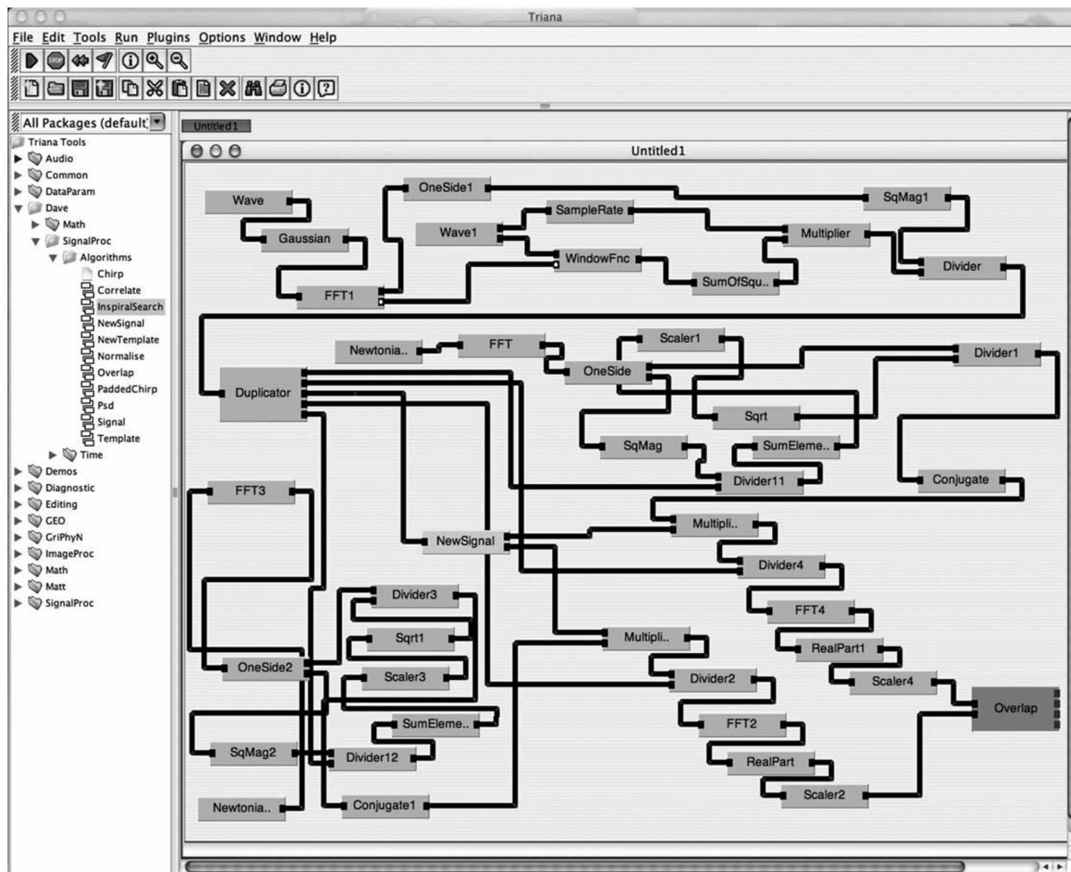

Figure 8. The Coalescing Binary search showing most of the units displayed on one work-space. Some units, e.g., Overlap, are themselves group units containing further task-graphs.

then the generated tool has two input nodes. The Triana units generated from discovered/imported WSDL documents are inserted by Triana into the user's tool tree, alongside the available local tools. These Web Services can thereafter be used as standard Triana units.

Invocation of Web Services is currently done using the Web Services Invocation Framework [34]. When data is sent from a local Triana unit to a Web Service, the data from each input cable for that service is packaged into a WSIF input message, and any data type conversions (e.g., string to double) are achieved. The Web Service is then invoked with the input message, and the data returned by the Web Service is passed to the next tool in the workflow along the relevant output cable. If the next tool is a Web Service then the return data is used to invoke that service, allowing Triana to choreograph workflows involving multiple Web Services. This feature combined with the BPEL4WS reader that we are developing will allow us to import and choreograph BPEL4WS workflows.

5.4. *Future Work*

Currently, Triana is being applied to implement the inspiral search workflow on a number of resources distributed across Europe within the GridLab testbed. We consider the following issues to be of significance in order to fully support a functioning testbed. These issues are currently being addressed and will be considered significant additions to work already presented here:

Data Management – the data will be stored in a decentralised fashion across the GridLab testbed and interfaced through the GridLab data management services. The data management team have already replaced the communication layer of the gravitational wave IO library for reading/writing data (called the Frame library) which allows geographical transparency by allowing local and remote access to be treated identically using a logical file reference. The user provides a GPS time as the

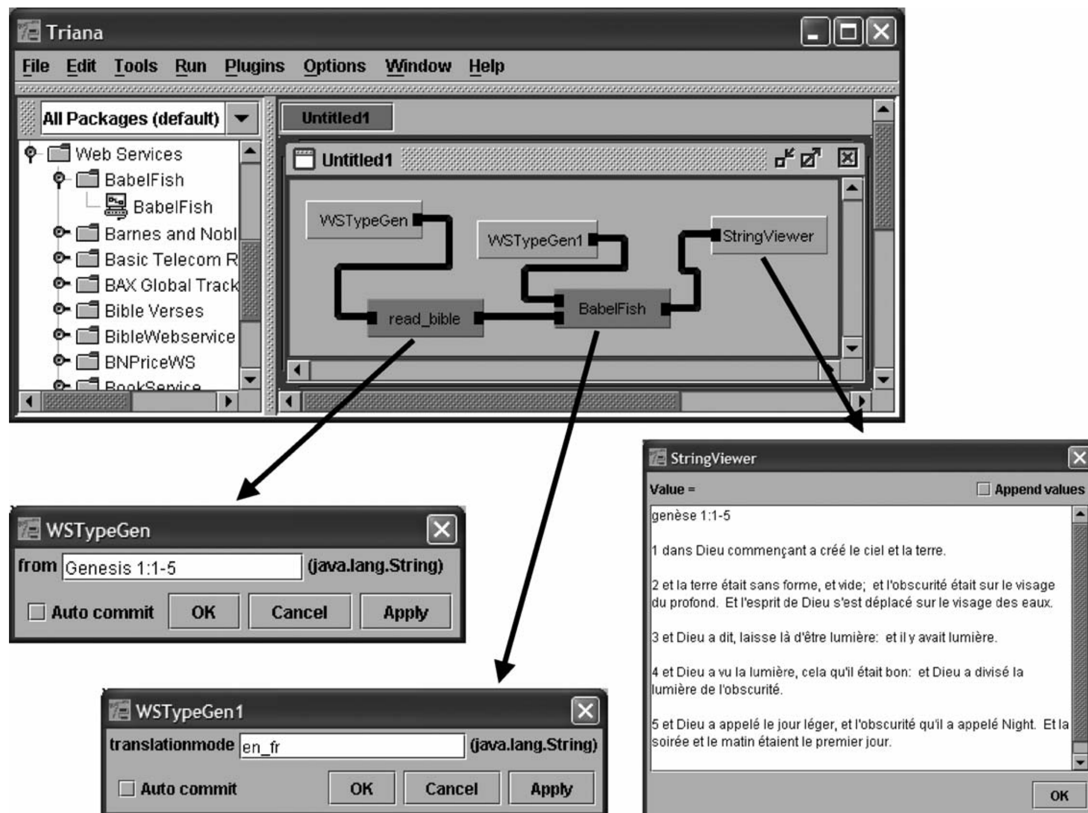

Figure 9. Web Services Workflow in Triana.

logical filename, which is converted into the file location on the set of distributed resources.

Security – currently, security considerations are not built into the GAP interface (unlike the GAT). We intend to integrate the Grid Security Interface based on X.509 certificates into the GAP. This will enabling us to contact the secure GridLab services that are currently available to us and deployed on the GridLab testbed.

Job Submission – based on the security infrastructure, we will be able to use GridLab services from within a Triana workflow in the same way that we can invoke Web Services now. This will not only allow us to connect to the GridLab GRMS service (for Globus-based job submission) but it will allow us to choreograph job submission workflow for complex submissions, e.g., job submission could involve a number of steps: CVS checkout, compilation, service deployment etc.

These services will be integrated at the GAP level and therefore extend its current functionality into a broader set of Grid services for application integration.

6. Conclusion

The use of Triana as a problem solving and composition environment for Grid-based resources is demonstrated. Triana enables users to compose applications based on workflow principles, allowing a variety of user developed and third party services to be integrated. Services may be executed on local or remote resources, parallel distribution of services is also supported. By isolating the mechanisms of distribution, via a collection of interfaces we call *upperware*, from implementation technologies, generally called *middleware*, we enable a variety of distributed infrastructures to co-exist. This has been a significant motivation for our work, and we demonstrate distribution mechanisms based on Web Services and Peer-to-Peer middleware technologies. Both the generic API and subsequent implementation of this are discussed.

As Grid computing matures, and embraces commercial-grade infrastructure (such as Web Services), it is essential that a variety of implementations for service management and execution co-exist. It is unlikely that all user communities will converge on

one technology for deploying applications – this has been a significant focus and motivation for this work.

Triana is now available as an open source software package, and includes the Galaxy formation visualization example, and the distributed Triana prototype. The current download includes the P2PS and Web Services binding and can be downloaded from the website [16].

Acknowledgements

We would like to thank both application groups mentioned in this paper: in particular, Roger Philp for supplying the motivating problem for the design of the distributed data example, advice and collaborative componentisation of his galaxy formation code; and to Dave Churches for his extensive proof-of-concept Triana network for real-world data analysis. Also, thank you to B. Sathyaprakash, Bernard Schutz and the rest of the GEO 600 team for their invaluable advice and input into the design and implementation of the current Triana system and data analysis tools. We are also indebted to the many members of the GridLab project for providing the vision for the Grid Application Toolkit (GAT) described in this paper. Finally, thanks to the Web Services implementation team including Shalil Majithia, Andrew Harrison and Diem Lam.

References

1. G. Allen, D. Angulo, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, M. Russell, T. Radke, E. Seidel, J. Shalf, and I. Taylor, “GridLab: Enabling Applications on the Grid: A Progress Report”, in *3rd International Workshop on Grid Computing (GRID 2002)*, held in conjunction with Supercomputing 2002, Lecture Notes in Computer Science, Vol. 2536, pp. 39–45.
2. G. Allen, K. Davis, K.N. Dolkas, N.D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, “Enabling Applications on the Grid: A GridLab Overview”, *International Journal of High Performance Computing Applications*, Vol. 17, No. 4, pp. 449–466, 2003.
3. BPEL4WS, “Business Process Execution Language for Web Services”, Version 1.1 05 May 2003. See http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.
4. O. Bunin, Y. Guo, and J. Darlington, “Design of Problem-Solving Environment for Contingent Claim Valuation”, in *Proceedings of EuroPar*, Lecture Notes in Computer Science, Vol. 2150, Springer-Verlag, 2001.
5. D. Churches, M. Shields, I. Taylor, and I. Wang, “A Parallel Implementation of the Inspiral Search Algorithm using Triana”, in *Proc. of UK eScience All Hands Meeting*, Nottingham, Sept. 2–4, 2003.
6. S. Fleeter, E. Houstis, J. Rice, C. Zhou, and A. Catlin, “GasTurbnLab: A Problem Solving Environment for Simulating Gas Turbines”, in *Proceedings of 16th IMACS World Congress*, 2000, pp. 104–105.
7. I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.
8. I. Foster, C. Kesselman, J. Nick, and S. Tuecke, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration”, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.
9. I. Foster and A. Iamnitchi, “On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing”, in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS ’03)*, 2003.
10. G. Fox, D. Gannon, and M. Thomas, “A Summary of Grid Computing Environments”, *Concurrency and Computation: Practice and Experience* (Special Issue), 2003. Available at: http://communitygrids.iu.edu/cglpubs.html.
11. E. Gallopoulos, E.N. Houstis, and J.R. Rice, “Computer as Thinker/Doer: Problem-Solving Environments for Computational Science”, *IEEE Computational Science and Engineering*, Vol. 1, No. 2, 1994.
12. GEO 600 Gravitational Wave Project Home Page, see http://www.geo600.uni-hannover.de/.
13. Gnutella. Gnutella File Sharing Network, see website http://gnutella.wego.com/.
14. Grace Development Team. Grace Grapher, see website http://plasma-gate.weizmann.ac.il/Grace/.
15. GridLab Project, see website http://www.gridlab.org.
16. GridOneD Project and the Triana Software Environment, see websites http://gridoned.org and http://www.trianacode.org.
17. IBM and Globus. IBM and Globus announce Open Grid Services for Commercial Computing, see website http://www.ibm.com/news/be/en/2002/02/211.html.
18. Jini, see website http://www.jini.org/.
19. M. Li, O.F. Rana, D.W. Walker, M. Shields, and Y. Huang, “Component-based Problem Solving Environments for Computational Science”, in Kung-Kiu Lau (ed.), *Component-based Software Development*, World Scientific, 2003.
20. M. Li and M.A. Baker, “A Review of Grid Portal Technology”, in J. Cunha and O.F. Rana (eds.), *Grid Computing: Software Environment and Tools*, Springer-Verlag, 2004, to appear.
21. S. Majithia, I. Taylor, M. Shields, and I. Wang, “Triana as a Graphical Web Services Composition Toolkit,” in *Proc. of UK eScience All Hands Meeting*, Nottingham, Sept. 2–4, 2003.
22. NOAA/PMEL/EPIC group. The Scientific Graphics Toolkit, see website http://www.epic.noaa.gov/java/sgt/index.html.
23. J. Novotny, M. Russell, and O. Wehrens “GridSphere: A Portal Framework for Building Collaborations”, 1st International Workshop on Middleware for Grid Computing (at ACM/IFIP/USENIX Middleware 2003), Rio de Janeiro, Brazil, June 2003. See Web site at: http://www.gridsphere.org/. Last visited: January 2004.
24. Open Grid Services Infrastructure, see website http://www.gridforum.org/ogsi-wg/.
25. Project JXTA, see website http://www.jxta.org/.
26. SETI@Home, see website http://setiathome.ssl.berkeley.edu/.
27. I. Taylor, M. Shields, and I. Wang, *Grid Resource Management*, J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinski (eds.), Kluwer, June 2003.
28. I. Taylor, M. Shields, I. Wang, and R. Philp, “Grid Enabling Applications Using Triana”, Workshop on Grid Applications and Programming Tools, June 25, 2003, Seattle. In conjunction with GGF8 jointly organized by: GGF Applications and Testbeds Research Group (APPS-RG) and GGF User Program Development Tools Research Group (UPDT-RG).

29. I.J. Taylor, R. Philp, O.F. Rana, M. Shields, and I. Wang, “Supporting Peer-2-Peer Interactions in the Consumer Grid”, in *Proceedings of HIPS Workshop at IPDPS*, April 2003.
30. I. Taylor, M. Shields, I. Wang, and R. Philp, “Distributed P2P Computing within Triana: A Galaxy Visualization Test Case”, Proceedings of IPDPS 2003, April 22–26, 2003, IEEE CD-ROM.
31. The Globus Project, see website http://www.globus.org/.
32. UDDI.org UDDI Technical White Paper UDDI.org, September 6, 2000, see website http://www.uddi.org.
33. W3C Web Services Description Language (WSDL) 1.1 W3C Note, March 15, 2001, see website http://www.w3.org/TR/wsdl.
34. Web Services Invocation Framework (WSIF), see website http://ws.apache.org/wsif/.
35. XMethods.com. A “Virtual Laboratory” for Web Services Developers, see website http://www.xmethods.com.