Published in Java/Jini Technologies II (part of ITCom 2002), Proceedings of SPIE, Volume 4863.

August 2002 in Boston, MA, USA.

Service Oriented Visualisations

Claire Knight[†] and Malcolm Munro

Visualisation Research Group,
Research Institute in Software Evolution.
Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.

ABSTRACT

It is important to be able to provide a variety of ways of visualising any data set if the differences between users in terms of ability and preferences are to be successfully overcome. Human variability is a wonderful thing; except perhaps when there is a visualisation being evaluated! There is not likely to be a representation that fulfils the needs of any given group of users and still remains likeable to them all. This provided the impetus to create a visualisation framework that is able to a variety of visualisations and data through a service mechanism. This allows for matching services to be located, and utilised, as necessary, using a contain client application that acts to hold the component parts of the framework together. This provides representations each user is happiest with to be employed (of those made available), and also for a comprehensive analysis and understanding tool because of the multiple sources of data that can be integrated. In order to best exploit the possible distributed nature of visualisations and data sources, and to provide extensibility for collaboration and Grid integration the implementation makes use of Jini.

Keywords: Jini, Visualization, Distributed Applications, Components

1. INTRODUCTION

This paper describes a visualisation framework and brokering system. It is by nature a distributed system that allows for a flexible and as-needed approach to the use of data converters and visualisation displays/tools/styles. The framework is known as VIBRO (Visualisation Broker Framework). It is designed to make use of the facilities that Jini provides in allowing distributed applications to be written without having to work at the network layer. The concepts of services, lookups, and clients are central to the idea of an extensible and flexible visualisation framework.

The rest of this paper provides some rationale for this work before defining the aims. A discussion of the approach is then presented with some information about the definition of the clients and services that constitute VIBRO. This defines some of the interfaces and behaviour expected of them. Finally a summary is given with some of the issues of using Jini in this way examined.

2. RATIONALE

Having flexible and customisable visualisations greatly improves their usability. This is because it allows each user to have the visualisations that they prefer working with operating on the data to be examined[2]. Trying to incorporate all of this variability in a monolithic application would require a large investment of time and resources for one site/team. It is also then not easily extensible when new data sources and/or presentation styles are developed. The solution therefore

[†] Correspondence should be directed to the first author. E-mail: C.R.Knight@durham.ac.uk. Tel: +44 191 374 2554.

seems to be to use a plug-in type application with support for new services. This can be achieved, as shown in this paper, through the use of technologies such as Jini.

2.1 Research

There has been similar research and development from the point of view of plug-in visualisations. Apart from the obvious plug-in components, which are sold for developers, there are three main related initiatives. The first is VTK[1], the second is Snap[3], and the third is compositional visualisations based on modules[5]. VTK (The Visualization ToolKit) is an open source software system that enables the creation of 3D graphics and visualisation through C++, Tcl/Tk, Java, and Python. It can be used on most Unix platforms and on Windows and is based around an object model for constructing visualisations. It does this by abstracting away from the graphics library level such as OpenGL. It also supports a wide variety of algorithms and modelling techniques. Because of the use of the object model, there is a collection of smaller units with well defined interfaces that allow easy of assembly into larger programmed systems.

Snap-Together Visualization is a model that creates explicit interaction links between multiple views, and is based on relations and actions. Interactions with objects in one view cause the views at the other end of the relationships to also look that data up based on the action defined for that link, and thus changing an object in one view will update all linked views. The compositional approach is a model for constructing complex visualisation instances from connected smaller modules. The smaller modules are themselves visualisations and represent some smaller part of the data. This work is similar to that by Roberts[6] on the use of multiple views and multiform visualisations. The *Compositional Model* describes visualisation instances as clustered graphs with the data as clustered directional graphs. The model is able to support the combination of visualisation modules into larger visualisation models (composition) and also navigation between the models.

One visualisation system that has been developed using similar technology is the collaborative visualisation system for military planning[4]. This uses a lookup service for producers to advertise that they have certain information available. Information consumers can then use this information. The paper discusses two types of consumers; fuselets and presentation services. The fuselets integrate various pieces of information, in effect tailoring the data, which is then re-published for other consumers. The presentation services display information to the user. Collaboration is achieved through presentation services that make use of a common data space.

2.2 Intention

There is the danger that given a brief look at this work that the view will be taken that this is purely the use of component technologies. This is not so, and it is not the intention to try and claim that this is research within the component arena. What is new and novel about this work is that it seeks to provide a way to create flexible visualisations. Because of the need to visualise program code and running instances for tasks like program comprehension there is also a need to support this process in a distributed manner. There is also a future requirement that data visualisation may need such support. There is very much a move towards the concept of service provision and of being able to work anywhere, anytime, and on any device. If visualisations (apart from obvious restriction such as display) are not able to support this process then it will take some time before they are then considered as an important part of any application or data service.

The aim of this work is to provide a visualisation arena containing a variety of visualisations, judged as appropriate from a visualisation perspective, that can be linked to various data sources as required by the user. This work is novel in that it seeks to automate this process through data brokerage, and to provide a distributed, customisable, and "pluggable" environment in which such visualisations can be used. It does not seek to further component research, but to utilise the techniques that have come from this area. Live data links can be utilised, themselves possibly as remote as the various visualisations. This allows for great flexibility in the pursuit of understanding, and also paves the way for such tools and sub-tools to be provided on a service/as-needed basis, as indeed is the way in which software delivery is seen to be heading.

Because of the plugin style approach adopted for this research, and moving towards using mediation and brokering between components many different visualisations can be used. These can differ in the metaphors and representations used, the number of dimensions employed for the graphics, and the type of data that they are suited for. Many different

sources of data can also be integrated, either with the same visualisation style or each using its own suitable graphical display.

An instant benefit of using other data sources and visualisations within an umbrella environment is the ability it gives to the user to be able to annotate their findings and knowledge, and to link this across views and data sources. Often this has only been supported within each visualisation (if at all), and whilst specific annotations directly onto the graphics may be desirable much of the time it is not. An area of future work may indeed be to interact directly with the visualisation in order to support this as well. This would also lead towards successfully creating linked windows, so that for any data source item, it could be located in any of the registered graphics currently displaying that data.

2.3 Use of XML

Apart from the use of Java and Jini for implementation, the framework also utilises XML for communication. There are various reasons as to why XML has been chosen and this section discusses them. One of the most obvious reasons, and often given by supporters of XML, is that XML tells you what kind of data you have, not how to display it. Because the tags identify the information and break up the data into parts, a variety of applications, services, and so forth can process it. Because the different parts of the information have been identified, they can be used in different ways by different services as necessary. It is also a benefit when it comes to generating information that a service then makes available to other services. A data service and filter pairing can be developed for most data and this then allows it to be used by any other service that can deal with the output formats of the filter service.

An additional benefit is that XSLT can be used to automate to some degree the transformation of the "raw" XML received by a service into only those tags that it wishes to deal with. Another strong advantage of using XML rather then byte streams or text files (and similar mechanisms) is that the content can be validated using DTDs or Schemas. As long as the validating specification is provided with the source XML then any code that can handle XML can at least read the file.

The main benefit of working with XML is that it is emerging as the standard for information and data exchange between (distributed) applications and services. As was identified in the conclusions of [9] there is a need to be able to integrate visualisations into everyday applications. Through the utilisation of XML the transfer of data necessary to the visualisations then just becomes another of the data consumers in the wider system.

2.4 Summary

It is important to understand that the VIBRO framework does not directly deal with any aspect of the low level graphics or any rendering algorithms. The visualisation services provide this facility to the client, and whilst the interface may be running on a local virtual machine (at the client) the generation of the appropriate images and components is done by the service. Any customisation of a visualisation service is also working through the proxy to the back end implementation of the service. Anything that is customisable by the user should be presented as part of the service configuration (user not administration level) to make this clear to users of the VIBRO client.

The main difference between this work and that presented in [6] is that this focuses on the integration of data and visual components at user command. The visualisations are themselves services and have to provide the same interfaces as any other service. Collaboration is also not a primary concern. Future work on this is to also consider more sophisticated brokering of data to enable visualisation needs to be met regardless of data source. This is not a case of looking for matching services, but of data conversion and visual mapping adjustment. Automation of this (through autonomous services) would be highly desirable. There is also much less focus on speed of response. It is the intention to create a highly flexible system that allows for comprehensive long-term investigation with persistence of analysis findings. An early version of this work, before the introduction of Jini and a refinement of the operational semantics, can be found in [7].

3. AIMS

The aim of this work is to be able to provide an infrastructure into which different visualisations and different data source can be integrated to give the widest possible choices for the analysis and understanding of that data. Much of the driving influences behind this work have stemmed from the task of visualising software for the purposes of program

comprehension, but the aims and the ideals can be extrapolated to most of information and database visualisation. It may even be reasonable to apply the ideas to scientific visualisation. Through the distributed nature of the framework and the growth of Grid computing in the scientific community, not only can the analysis tools be distributed but also so can the processing. There is also the added impetus of being able to utilise such work within knowledge discovery and database fields as the use of visualisation for both helpful "auxiliary displays" and also the actual process of knowledge discovery and refinement is now starting to be considered valid.

More detailed objectives can be considered to be:

- To be able to select the data to be visualised
- To be able to refine any data selection that is currently being visualised and to then see the visualisation reflect this refinement
- To be able to control the visualisation displays by configuring colour etc.
- To have a choice over the interaction used to work with any given data
- To have a choice over the displays used to present any given data
- To be able to support a variety of views of the same dataset at the same time

The final three of these objectives are essentially where the distributed and pluggable nature of the framework comes to the fore. Without some form of extensibility then the visualisations available are only those created as part of the application. The data styles would also be limited other then rewriting other large chunks of the application. By utilising a service nature for these parts of the application the physical location of the code becomes irrelevant, and the flexibility and choice increases.

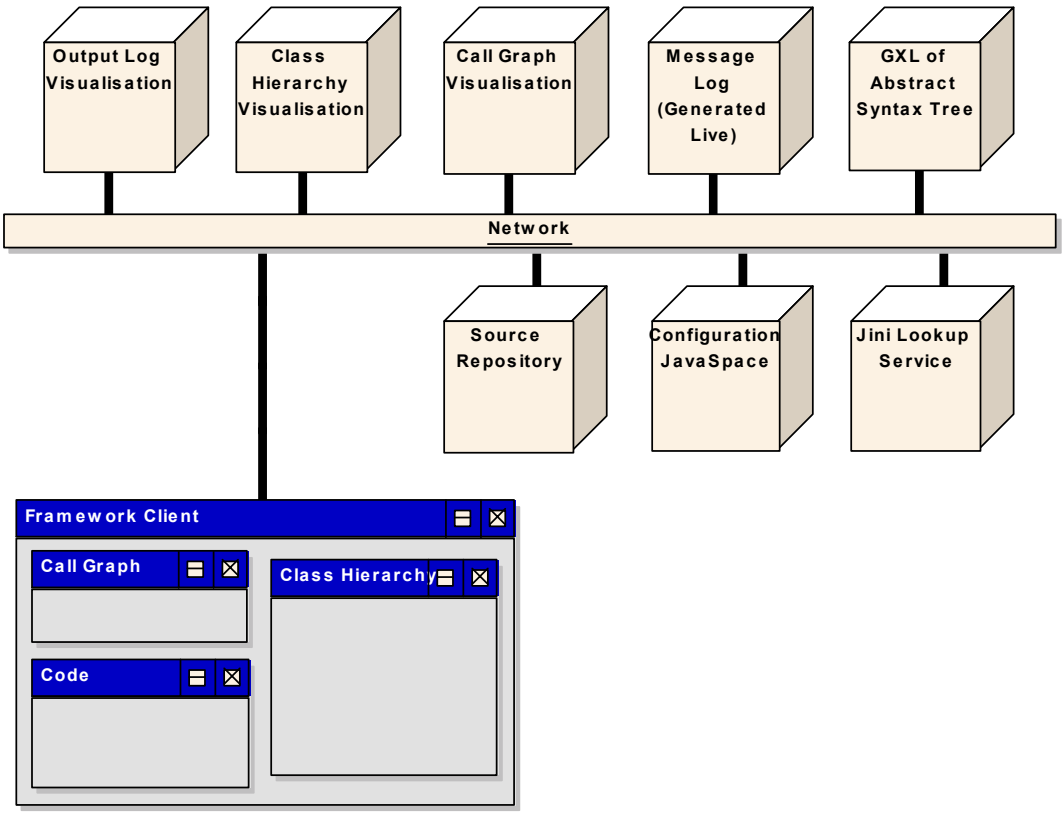

Figure 1 - Theoretical Realisation (Software Visualisation Example)

4. APPROACH

The framework for this distributed visualisation and analysis solution essentially acts as the Jini client. Using the lookup and discovery notions, the client is able to locate and utilise any data or visualisation service (see the next section) available within the federation. This allows for a flexible approach to the composition of views of the data and facilitates comprehensive and personalised understanding. This is through the amount of choice the user has over presentation and display. Figure 1 displays a theoretical instantiation of this framework with one client and several services on the network, whilst Figure 2 shows the overall system design.

4.1 Overview

This paper provides an introduction to a visualisation framework and brokering system. It is by nature a distributed system that allows for a flexible and as-needed approach to the use of data converters and visualisation displays/tools/styles.

Because of the wide range of visualisations and data sources (services or Jini services) that can be utilised within this framework, then it is necessary to provide some type of container through which they can be integrated and actually used. The client application fulfils this role, but also provides greater functionality such as working at the data description level to determine which data sources are suitable for linking with which visualisations. The links between the services are also handled at this level. Once the links have been made the event model used within the framework allows for the data and visualisation sources to communicate.

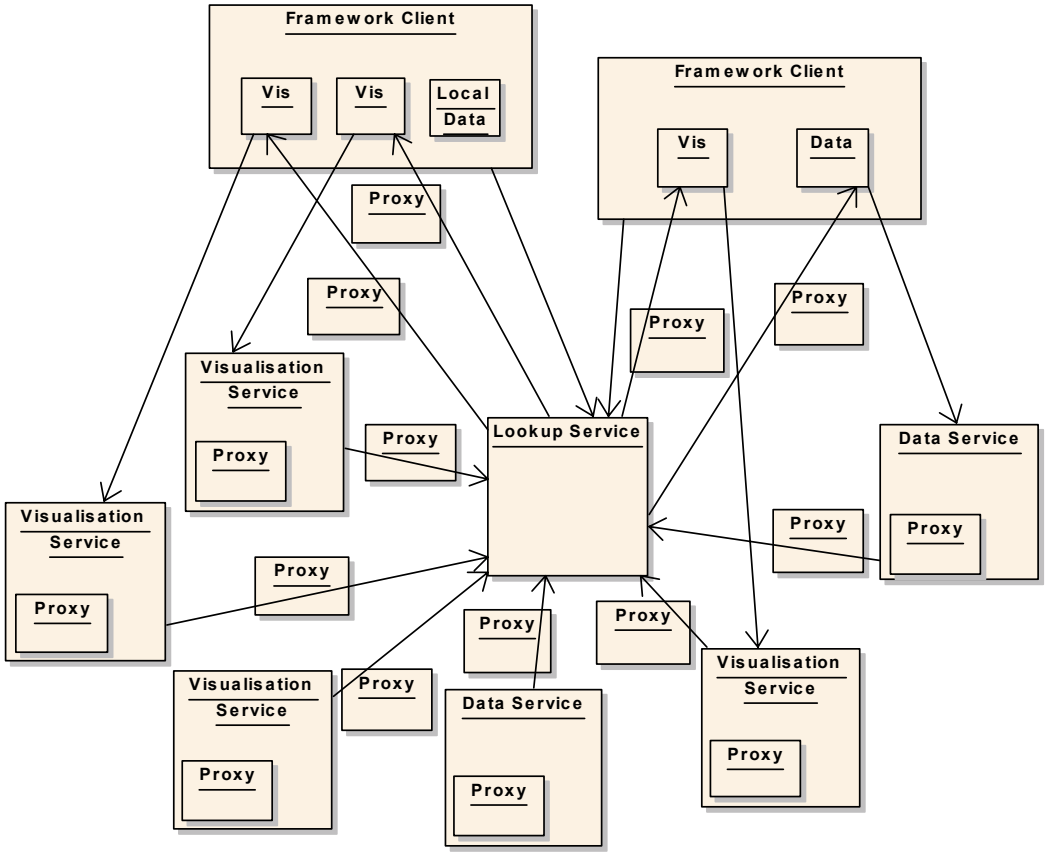

Figure 2 - Overall System View

Any visualisation service (apart from adhering to the detailed specification below) must act as a self-contained unit. The use of metaphors, graphical organisation, layout, and interaction should all be part of that entity. There is no recourse to using graphics routines in the client application; there are none. If, in the name of reuse, there were some shared aspects of visualisation services that make sense to be modularised and reused then it would be advisable to make these also services. However, as far as VIBRO is concerned, these are just any other Jini service, and it makes no provision for working directly with them, nor ensuring their availability. The service(s) using them must do this.

A data service deals with the location and provision of the data that it represents, and may provide conversion services. The internal representation and storage (and whether it can support permanent change to that data, or which of the addition, deletion, change operations that are supported) are the responsibility of the entity. The actual physical location of that data is also entirely down to the function of the entity.

In addition to the data and visualisation services, there is a third type of service; filter services. These act as manipulation gateways through which the data can pass between the other services. This allows for a variety of transformations and filters to be applied to the data. The filter services take care of all the algorithmic needs of the service – essentially they act as a pipe by receiving data, acting on it in some way, and then passing it on to another.

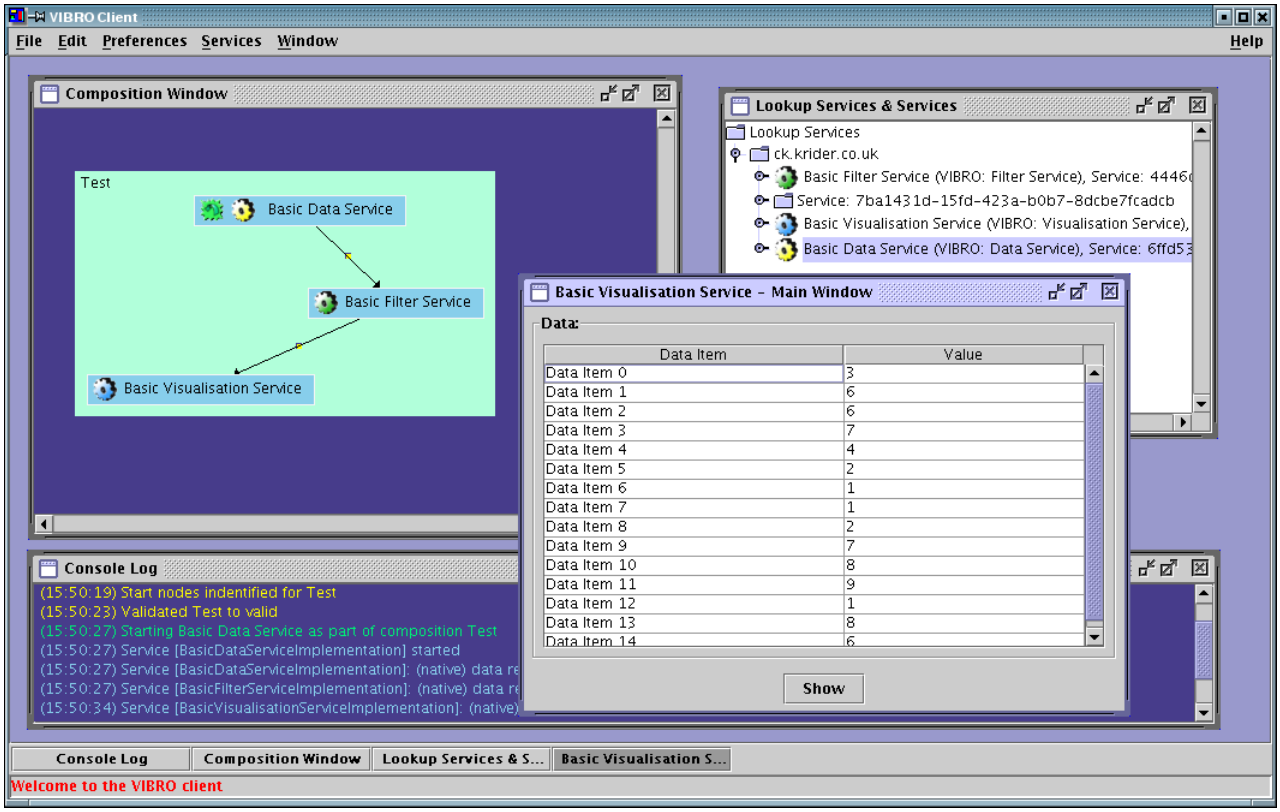

Figure 3 - VIBRO Client running on a Linux machine

4.2 Client

Because of the intended multiple service use of this client then the main interface is a multiple document interface (MDI). This is implemented in Java using the standard Swing API and implemented using JFrame, JDesktop, and JInternalFrame, and also through some specific subclasses that allow for specialist behaviour within the client such as a window button bar, and general window management from a menu. This means that the various interfaces returned by

the services to the clients can all be displayed at once as they can each be allocated to a JInternalFrame and integrated into the main interface. This can MDI can be seen illustrated conceptually in Figure 1 and realistically in Figure 3.

The client is essentially a thin client because all of the main processing is done by the services that it utilises. The client has to be able to locate and then obtain the necessary services, and to broadcast events between services that support this. It also has to be able to pass the required data to the visualisation services. And finally it has to act as the central interface to the whole process by providing the user interface into which the services are placed for display.

Figure 3 shows a screenshot of the client in action. This shows a variety of windows displayed within the standard VIBRO client. At the top of the screen there is the usual menu bar, whilst at the very bottom there is a status bar. This status bar can be used to display messages to the user from the client application. Just above the status bar there is a window toolbar that allows the current window to be selected. In this instance it is not really necessary but it is likely that during any reasonable realistic analysis there will be many windows open at the same time. This toolbar allows the user to better manage them.

Moving into the MDI desktop display of Figure 3, there are four windows. The service composition window in the top left, the service browser in the top right, the console running along the bottom, and overlaying all of these there is the main window of the *Basic Visualisation Service*. This GUI is provided from the service via factory methods and a `UIDescriptor` as per the ServiceUI[8] specification. This window has been shown by right clicking on the icon representing that service in the composition view and then choosing from the possible windows (in this case only the one). The population of the table is the result of the data held by the service, and in this case is the outcome of taking the original data from the *Basic Data Service* and then applying the *Basic Filter Service*. The console, service browser, and composition display are discussed in detail in the following sections.

4.2.1 Console Window

The console window provides a means for a variety of messages to be displayed to the user. It provides more flexibility than displaying messages via the client status bar, and also means that services and error handling code do not have to provide their own means of communicating to the user. It should be noted that only client-side service messages are displayed in this window. It would make little sense for server side messages to be sent to the client! Remote messages such as these are handled via the `RemoteEvent` mechanism of Jini.

Each message is preceded by the current system time when it is printed to the window. The messages are displayed in different colours to provide a quick identification of where the message originated. Figure 4 shows the different message types and the colours that they are displayed as, although the distinguishing colours will not be obvious in black and white. This output can also be obtained at any point by the user through right clicking on the console window to bring up the console menu, or by accessing the console submenu from the File menu. This contains an option to *Show Message Colors*, which produces the last five lines of output shown below as a reminder, or a key.

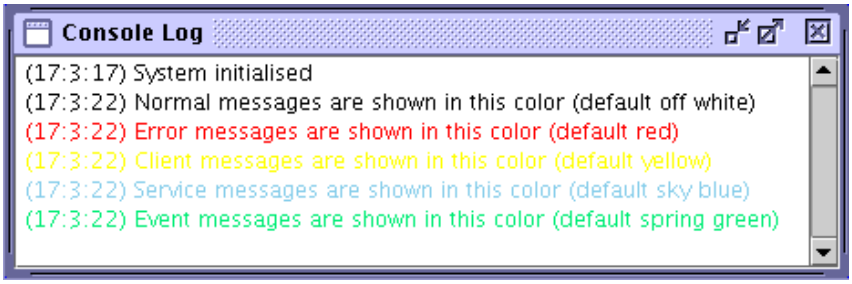

Figure 4 - Console window showing possible message colors

The console window in Figure 3 is showing several messages. The first two are related to client activity, as they are yellow in colour. They relate to composition activities. The next message is event based, and notifies the client that they have started this composition of services. In this case the event was fired from a service, although that is not always the case. The final four visible messages have been sent from the services involved in the compositions. This sort of output

is generally unnecessary but does provide some indication that things are going on in the background between the services. It is not a forced condition of services, but is useful, that here the message string includes the name of the service, which can be very useful to a user tracking activity.

4.2.2 Browser

The browser window is a means for users to view the services that are available to them. The browser is capable of displaying a wide variety of information about the services it finds. All lookup services are indexed by machine. Figure 5 shows a service browser in detail (in fact the one running in Figure 3 but with more branches of the tree opened). In this case there is one lookup service that is running with three VIBRO services registered with it. There is also the related service that is found by Jini clients listed as the second sub node of ck.krider.co.uk as Service: 4036aa2c-7195-4531-826b-ebab2652100c. This could have been filtered, but there is a design decision to list all services, but to make it easy to identify VIBRO services at a glance icons are displayed by the service name. Since there are three types of VIBRO service defined, there are three differently coloured but same image icons used for this purpose. Filter services are shown as green, visualisation services as blue, and data services as blue.

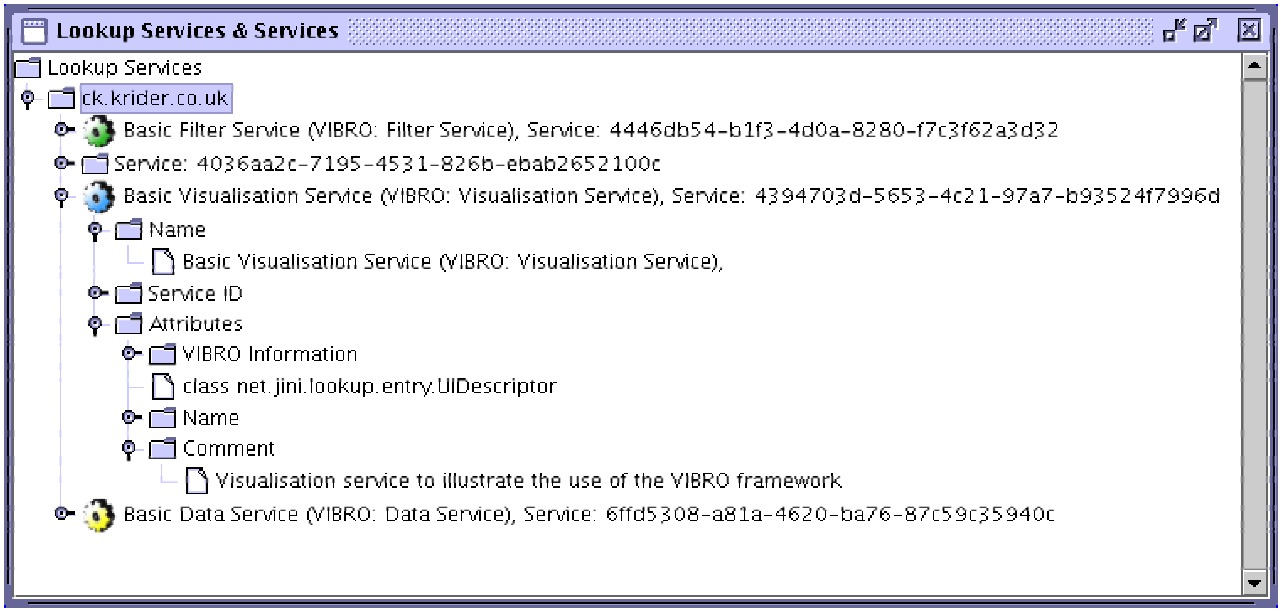

Figure 5 - Service browsing

In addition to identifying the VIBRO services, the browser allows a variety of attributes of a service to be displayed. The name of the service is taken from the definition of the service as being a VIBRO service. The service ID is also provided for those users who wish to track which instance of a service they are viewing and possibly using. The standard attributes of Jini are handled, and provided with a node that they can populate with their information. The VIBRO information is provided in more detail in here, as well as the attribute types defined as part of the current Jini API.

4.2.3 Composition Window

The composition window is where the real work of utilising the services takes place. The window supports drag and drop operations of services from the browsing window. Once a service has been dropped onto the window it is represented as an icon. If the service is not a VIBRO service it is shown as plainly with no icons, displaying only the service ID. However, the display is more informative if it is a VIBRO service. The service image used in the browser display, utilising the same colourings, is used before the human readable name of the service is displayed. A final image is used in the service icon if, once composed, that service is considered to be a start point of that composition. The client calculated the dependency is using graph analysis, and it then adds the icon to the necessary services. In the composition

window in Figure 3, there is one simple composition of services in serial. The arrows show the links that have been made between the services. There is one start point, and the composition has also been validated as it is shown with a background of green.

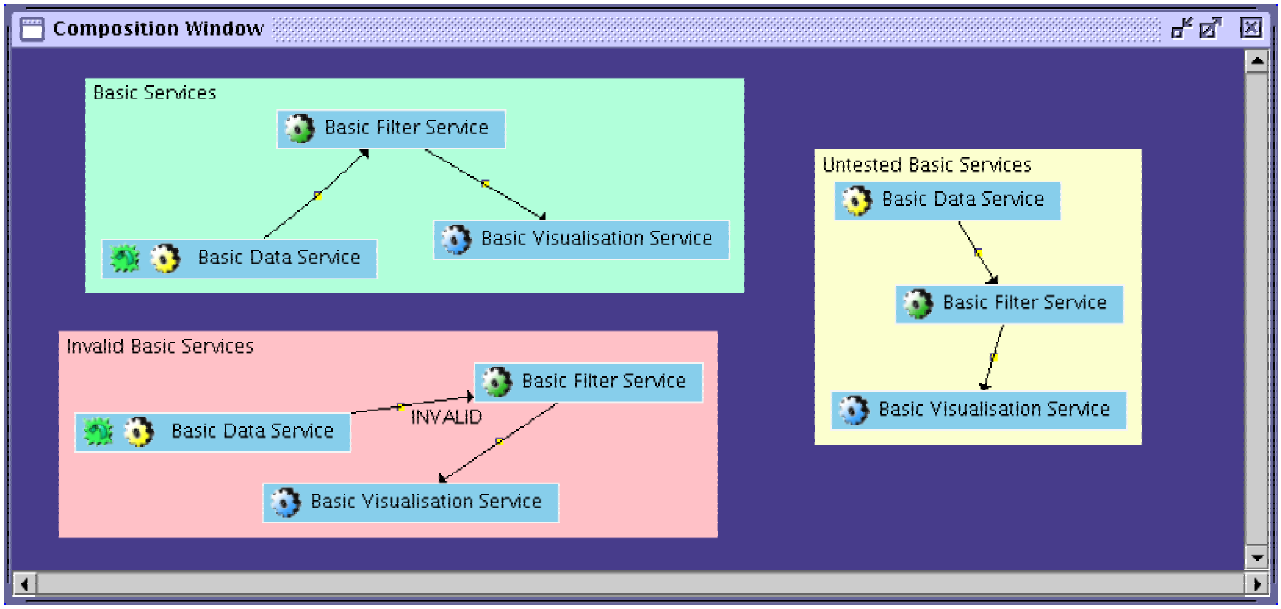

Figure 6 - Service composition

Figure 6 shows the three basic services provided as a demonstration and test for the API linked in three compositions. The top composition *Basic Services* shows the linked services with start node identified, and also validated. A pale green background illustrates this. The composition below this, *Invalid Basic Services*, shows the same composition but with one of the joining edges labelled INVALID. This is because the means by which the services communicate needs to be defined and match, based on the output formats (or DTDs/schemas) of one and the inputs of the other. An unvalidated composition cannot be run, and to be valid all links between services must be valid. The background colour of this composition is pale red. On the right there is a composition, *Untested Basic Services*, with no start points defined. It is coloured pale yellow, which is the default colour for compositions, as it has not yet been tested for validity. The background colours are pale because they have some level of transparency set so that overlapping compositions or services in multiple compositions can still be identified visually.

4.3 Services

There is a restriction of the sort of services that the client can utilise. Whilst the Jini concept allows true extensibility, it only makes sense for this client to use services that provide some form of data conversion or analysis services, or some visualisation services. To this end there are defined interfaces that these services must implement. These (at this moment in time) are essentially no more than placeholders for identifying appropriate services but have the potential to evolve. The use of a graphical user interface is central to the interface oriented view of this client application. This means that there is a requirement for services to provide a user interface in their attributes that is readily available. Because of this, the ServiceUI[8] concept has been adopted because of the standard interfaces that this specification defines. A VIBRO service has to adhere to these characteristics (i.e. they are essential):

- To be able to work as a standalone service. Even if this is not meaningful (for example a visualisation with no data) there should be default behaviour and displays and the service should not crash.
- Implement standard VIBRO interfaces so that clients can locate and connect to it
- Have a client user interface implemented and provided through the use of service attributes
- Implement `VIBROInfo` as one of the attributes of the service.
- Be a well behaved Jini service with respect to join protocols, lease management, and so forth

- Distributed as two jar files; one for the client code (downloaded) and another for the service back end.

Any visualisation service must act as a self-contained unit. The use of metaphors, graphical organisation, layout, and interaction should all be part of that entity. There is no recourse to using graphics routines in the client application; there are none. A data service deals with the location and provision of the data that it represents, and may provide conversion services. The internal representation and storage (and whether it can support permanent change to that data, or which of the addition, deletion, change operations that are supported) are the responsibility of the entity. The actual physical location of that data is also entirely down to the function of the entity.

According to the Jini Specification[10], there are several implementation guidelines that must be followed for a service to be considered a well behaved, or as it has been termed, a good citizen. The services that make up VIBRO or are implemented against the VIBRO interfaces must also adhere to these guidelines. Essentially a VIBRO service should:

- Upon starting, a service must discover lookup services of appropriate groups and then register with (join) any that reply.
- Whilst running, services must listen for lookup service startup events and if the new lookup services are appropriate to the group registering with them.
- Each service must "remember" its join configuration; the list of groups it should join and the lookup locators for specific lookup services. This information should be made persistent in some form locally to the service implementation.
- Each service must also "remember" all attributes that it has, and inform all lookup services of which it is a member of any change in those attributes.
- Leases with the lookup services must be maintained for as long as the service is available (notwithstanding network problems, crashes, removals of lookup services, and so forth).
- "Remember" the service ID assigned to the service by the first lookup service on its first registration so that all registrations of the same service wherever and whenever made, will be under the same service ID.
- Provide an instantiation of `uk.co.krider.vibro.services.entry.VIBROInfo` as one (or the only) attribute of the service. The proxy used should implement either directly, or indirectly one of `uk.co.krider.vibro.services.interfaces.VIBRODataServiceProxy`, `VIBROFilterServiceProxy`, or `VIBROVisualisationServiceProxy`. These all extend from `VIBROServiceProxy` in the same package. There are adapter classes provided for convenience and there are also backend service interfaces that the service backend should implement if the adapter classes are to be used. This allows for the VIBRO client and any other VIBRO service to know some basic information about the service, and also to make assumptions about the interfaces available.
- The provision of user interfaces is dependent on the service, but this should be done in the way that is advocated in the ServiceUI[8] specification. The VIBRO client expects a DesktopInternalJFrame window handle and therefore provides a factory in `uk.co.krider.vibro.ui.serviceUI.factory.DesktopInternalJFrameFactory` that can be used with the UIDescriptor classes that is provided as a service attribute.

5. ISSUES

There are many unsolved issues related to this work. Some of these are problems that are related to the visualisation theory that the services and clients are being used for, whilst others relate to implementation concerns. There are also some areas in which a variety of possibilities are feasible at a theoretical level but to be implemented, used, and illustrate the concepts, require that a solution is found that can be successfully implemented with the current reference implementation.

5. 1 Data

There are several issues relating to the data provided by services. One of the main ones is what are the benefits of having data distributed with the services, or is this data in some other arbitrary location that is only the concern of the data service? In this case the data is never seen other than through proxy access. The visualisation, along with its abstractions, is all that is seen by the user. If a data service then does not provide an interface for viewing some or all of the data there

is no user knowledge as to its content (other than via descriptions), its completeness or its validity. This can be exploited with regard to security, but can be a disadvantage for thorough analysis. Does this provide real security over the data, whilst still providing some analysis? How would a service work if it had its initial data locally to the client? The visualisation of remote data is also an area in which future work would be interesting. It may be that this system can act as some form of visual monitoring agent, through the services taking data local to it but remote to the client.

There are many possible answers to the above questions, but for the moment the decisions made are to delegate responsibility to the data service. The ideas and issues provided above provide an interesting way in which to expand this work and to investigate which solutions work best in different situations. In the current reference implementation, a data entity deals with the location and provision of the data that it represents. The internal representation and storage (and whether it can support permanent change to that data, or which of the addition, deletion, change operations that are supported) are the responsibility of the entity. The actual physical location of that data is also entirely down to the function of the entity. If a program comprehension data source providing dynamic information of a Java class executing in a virtual machine requires the use of introspection (for example) to extract these values from a remote server, then it needs to make its own provision for dealing with this.

5.2 JavaSpaces

There are two main uses of JavaSpaces that could be considered for integration into VIBRO. The first is for the storage of configuration information, and the other is for moving code around. Using JavaSpaces for the exchange of executable content is a convenient means of extending running code. While in the space, objects are just data, but once read or taken then a local copy is created. As with any local object, it is possible to invoke its methods, even if it has never been seen by the current application. This capability is a powerful way of extending the behaviour of VIBRO. Whilst this approach is attractive in many respects, the complexity of services being developed leads the authors to believe that it is unfeasible at the moment to approach VIBRO service provision in this way.

One use of a configuration oriented JavaSpaces is to store configuration for user-service pairs, so that if a user then uses another machine with the same visualisation services they can retrieve their settings. It is more realistic to perhaps consider the storage of compositions as complete entities and then store the user-entity pair in the JavaSpace because this is the conceptual level at which the client operates in terms of loading and saving. There should also be the ability to save client properties to the space. This means that whichever client installation was being used, as long as the user name can be located, the preferences of that user can be restored from and then resaved to the space. Currently these are saved to local disk using `java.util.prefs.Preferences` and then explicitly written/read using streams. Using the `Preferences` implementation means that future use of JavaSpaces will be easy to accommodate.

Obviously the location of the JavaSpace used for configuration is important. Clients should perhaps have certain servers on which they should look for the configuration space. There is also a need for replication so that there is not a single point of failure in the network system. The client should still be usable, even if the old configuration cannot be retrieved, using the defaults. There then needs to be a means of reconciling information on re-saving these configuration options and replacing or merging with any old ones in the space.

5.3 Graphics

It will also be interesting to note the effect the bandwidth has on the visualisations. Initial use is in an Intranet environment, but as part of a large university network, traffic problems can and do occur. The visualisation service design is based around a separation of concerns for exactly this reason. The interfaces have to access the service via the proxy. In addition to the standard data transfer methods for moving data between services, the visualisation service has to have a means for notifying the user interface that the data has changed and requesting it to update itself. The reverse communication is also necessary when the interface is used to request more data.

This means that the proxy must be expressive enough to deal with the user interface requests, and deal with the state information required to ensure that views are not synchronized between all users of the service. The proxy also has to filter events that are meaningful for the current user view if these are at the interaction level. It also requires that the

interfaces provided to the client are capable of dealing with the visual display independently of the service, apart from the actual data. There are two main issues that remain to be seen, and possibly investigated further, with more use of VIBRO. The first is of serialising graphics via the interfaces, and the second is of distributed graphics having lots of latency in interaction.

6. SUMMARY

This paper has shown some of the detail of a Jini based visualisation framework. The related literature has also been presented to provide background for this work. Figures 1 and 2 illustrate the physical and logical aspects of the system. Figures 3, 4, 5, and 6 then show the framework in action in the reference implementation of the client and the API through which other services interact with the framework. Sample services illustrate some of the potential of the implementation. Most of the effort has been on ensuring the interfaces are suitable for the intended use to which they will be put. This is to make certain that as much as possible has been designed to allow for the desired flexibility and expansion.

Because of the wide range of visualisations and data sources (Jini services) that can be utilised within this framework, then it is necessary to provide some type of container through which they can be integrated and actually used. The client application fulfils this role, but also provides greater functionality such as working at the data description level to determine which data sources are suitable for linking with which visualisations. The links between the services are also handled at this level. Once the links have been made the event model used within the framework (client) allows for the data and visualisation sources to communicate.

There is plenty of future research linked to this system. A reference implementation allows many of these to be explored. The availability of a well defined interface and runnable environment for visualisations will also help other visualisation research and new developments which require some form of system in which to run.

ACKNOWLEDGEMENTS

This work is partly financed by an EPSRC ROPA grant: VVSRE; **V**isualising **S**oftware in **V**irtual **R**eality **E**nvironments, and partly by research funds owned by Professor Malcolm Munro at the University of Durham. Thanks to Stuart Charters for a variety of conversations and constructively critical feedback during the development of this research and its reference implementation. Thanks also to Steven Glover and Andy Hatch for providing insight during discussions with the lead author.

REFERENCES

1. The Visualization Toolkit (VTK), available from http://www.kitware.com/vtk/vtkoverview.html.
2. **C. Knight**, *Visualisation Effectiveness*, Workshop on Fundamental Issues in Visualisation, in Proceedings of the International Conference on Imaging Science, Systems, and Technology (CISST), Las Vegas, June 2001.
3. **C. North** and **B. Shneiderman**, *Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata*, Advanced Visual Interfaces (AVI) 2000, May 2000.
4. **T. Jacobs** and **S. Butler**, *Collaborative Vlisualization for Military Planning*, in Java/Jini Technologies, Sudipto Ghosh (Editor), Proceedings of SPIE, Vol. 4521, pp42-51, 2001.
5. **R. Nagappan**, *A Compositional Model for Multidimensional Data Visualisation*, Visual Data Exploration and Analysis VIII, in Proceedings of SPIE, January 2001.
6. **J. C. Roberts**, *Multiple-View and Multiform Visualization*, Visual Data Exploration and Analysis VII, in Proceedings of SPIE, January 2000.
7. **C. Knight** and **M. Munro**, Mediating Diverse Visualisations for Comprehension, in Proceedings of the IEEE International Conference on Program Comprehsnion (IWPC), May 2001.
8. **B. Venners**, Jini ServiceUI Specification, available from http://www.artima.com/jini/serviceui
9. **S. Card**, **J. Mackinlay**, and **B. Schneiderman** (Editors), Readings in Information Visualization: Using Vision to Think, Morgan Kaufmann, February 1999.
10. Jini Specifications, avaibable from http://wwws.sun.com/software/jini/specs/.