Support for interactive heavyweight services

Julio López David O'Hallaron

Feb 15, 2001

CMU-CS-01-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Interactive heavyweight services are processes that make intensive use of resources such as computing power, memory, bandwidth and storage. The interactive nature of these services means that they have to satisfy user requests in a timely manner. An example of this kind of service is the remote visualization of massive scientific datasets. In order to present a visual representation of the dataset to the user, the visualization process has to execute a series of compute intensive transformation to the data. These heavyweight services usually encounter situations of *limited*, *heterogeneous* and *dynamic* resources. In order to deploy this type of service we propose a simple mechanism, called *active frames*, that easily enables us to move work and data to other compute hosts. With this mechanism, heavyweight services can aggregate resources from multiple compute hosts to satisfy a request. The active frame mechanism allows the use of application-level information in the selection of resources to satisfy the request. This is accomplished by delegating the resource selection to an application-level scheduler. Our initial results show that this flexible mechanism does not introduce significant overhead.

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9980063, and in part by a grant from the Intel Corporation.

Keywords: resource-aware applications, heavyweight services, active frames, internet services, remote visualization

Contents

1 Introduction

There exists a kind of Internet service that require vast amount of resources to satisfy a request from each single client. Examples of this class of service include datamining, advanced search engines with sophisticated information retrieval algorithms, distributed virtual reality environments and the remote visualization of large scientific datasets. To allow user interaction the service should satisfy the request and provide a response within a reasonable time limit. We call this type of services *Interactive Heavyweight Services* [9].

A challenge to deploy interactive heavyweight services is to allocate the appropriate set of resources needed to satisfy each request. Usually, the resource requirements of these services are not well understood or difficult to characterize and vary according to the parameters of each request. Moreover, these services usually execute in environments with *limited*, *heterogeneous* and *dynamic resources*.

Limited resources. Services execute on hosts with finite compute power. Server compute cycles are used to satisfy requests from multiple clients. For heavyweight service each request requires large amounts of computation resulting in the support of a relatively small number of clients.

For example, in a remote visualization service the datasets to visualize are stored at the remote site where the data was collected or generated through simulation. In an ideal world, we would simply copy the entire dataset from the remote super-computing center and manipulate it locally. Unfortunately, copying multi-terabyte files is not feasible in today's networks. Copying a 1 TB file over a 100 Mb Ethernet would take more than 22 hours assuming no cross traffic and no protocol overhead. Even if we had access to sufficient network bandwidth, we would not have the resources to store and backup multi-terabyte files. More important, the client would not have enough computing power to process the dataset and generate the desired visualizations. The crucial implications are that (1) large scientific datasets must reside on the remote site where they are computed; and (2) interactive visualization applications must be able to *aggregate* multiple resources across machines at the remote and local sites to manipulate these remote datasets.

Heterogeneous resources. Services execute in environments composed of heterogeneous resources. Networks have different bandwidth and machines have different computing power, memory, and display capabilities. For example, specialized workstations for visualization have a large amount of memory and accelerated hardware rendering engines, whereas regular desktop computers and laptops have relatively less compute power and modest graphics capabilities. Users want to utilize these services from the machines they have access to, using the resources available to them. It is desirable to be able to run the same visualization application across these different resource configurations, with different quality level according to the available resources.

Dynamic resources. Services often execute with shared computing and networking resources. These resources operate either under a *best-effort* model or a *reservation* model. In a best-effort model the amount of resources a service obtains to process a request is not guaranteed. Networks become more or less congested, processors become more or less loaded, and thus the availability of these resources changes over time. As more clients make simultaneous requests to a server the effective compute power and bandwidth to process each request decreases, thus increasing the delay experienced by the user. In a reservation model, there exists a resource manager that arbitrates the access to resources. Under this model the resource availability is less dynamic. Once the access to a resource is granted the requested amount of resource is guaranteed to be available for a negotiated time period. However, before negotiating with the resource manager the application must determine the amount of resources it needs to execute.

In order to deploy interactive heavyweight services, we need ways to deal with limited, heterogeneous and dynamic resources. Services need to aggregate resources when possible to deal with insufficient resources. Services need to be performance portable to can run under heterogeneous resource configurations. In general, services need to adapt to the available resources and degrade gracefully when there are not enough resources.

We propose a model to provide interactive heavyweight services on a *computational grid* [14] of hosts. In this model the service aggregates resources from the remote and local site as well as from within the network to satisfy a request. With this model users accessing the service can contribute additional resources to execute the service. Section 2 describes the resource model to deploy interactive heavyweight services on a computational grid.

For this model to be effective, the applications must be able to use the resource provided be these different hosts. We have implemented a simple mechanism, called *Active Frame*, that allows the application to easily move work and data to hosts in the computational grid. An active frame is an application-level transfer unit that contains both application data and a program. The program included in the frame specifies the computation to be performed on the data. Active frames are processed by active frame servers, or simply frame servers, that run in hosts located in the computational grid. The active frame mechanism is described in section 3. Section 3.3 explains how to create application-specific extensions to provide high-level services

We used the active frame mechanism to build a simple remote visualization service that we call Dv. This service enables the user to generate in his local workstation a visual representation of a dataset stored remotely. Section 4 presents an overview of the structure of the service. We created extensions to the active frame mechanism to build our remote visualization service. These extensions are a set of specialized active frames and application routines used to move and process the datasets in the compute hosts. We expect these extensions to be general enough to be used in similar remote visualization services. Section 6 describes these extensions.

To deploy our remote visualization service under different resource configurations we use application-level schedulers [5]. Section 7 describes the scheduling interface, which is a mechanism that enables the use of application-specific information in the resource selection process.

Section 8 resumes how the active frames mechanism is used to build the remote visualization service. This section also explains how the Dv extensions could be used to build a different visualization application.

The flexibility offered by the active frame mechanism could introduce potential overhead. Section 9 evaluates the cost of processing active frames in a remote visualization application. The application is evaluated in two different resource configurations. Our initial results show that the overhead introduced by the mechanism is reasonable.

The work reported here is a step towards supporting the creation of resource-aware applications. Section 10.3 points out the direction for future research. This work integrates different approaches developed by other research projects. Section 11 describes the related work.

2 Resource model

It is often the case that heavyweight services encounter limited resources to execute. In the model we propose, these services are executed on a computational grid of hosts [14]. Services aggregate resources from different hosts to obtain the required computational power, memory and satisfy other resource requirements. In this approach the application is partitioned into many computational modules, which in order to be effective, have to be placed intelligently in the available hosts. The static placement of the modules is not adequate in all cases. The appropriate partitioning of the application is both application and resource dependent.

Figure 1 ilustrates this resource model. Suppose that a scientist at some site has an interesting dataset that he wants to make available to members of the research community to visualize and otherwise manipulate. We will refer to him as the *service provider* and to his site as the *remote site*. As a part of creating the service, the service provider designates a collection of $m \geq 1$ hosts under his administrative control to satisfy service requests. At least one of these m hosts has access to the dataset and in general the m hosts will be shared with other applications. The m hosts will typically be physically located at the remote site, but in general

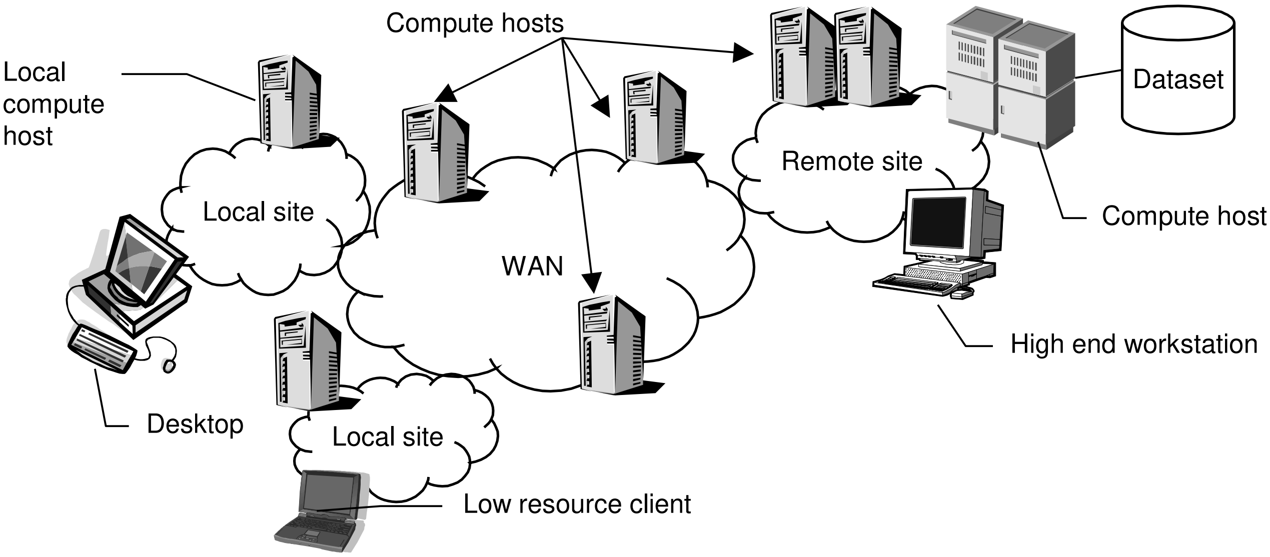

Figure 1: Grid resource model.

this is not a requirement; some of the m hosts could be located within the network and they are acquired using grid-based resource management services such as Globus [13].

A scientist at another site (i.e., a *service user* at the *local site*) visualizes or manipulates the remote dataset by issuing a finite series of requests to a host on the remote site. The period of time between the first request and the last request is called a *session*. Before the scientist begins a particular session, he must designate a collection of $n \geq 1$ hosts that are available for running any programs that are needed to satisfy service requests. One of this n hosts will be designated as the client host. The client hosts runs the processes that provide the interface for the user to interact with the remote service. As with the m remote hosts, the n hosts are not necessarily physically located at the local site, although most likely they will be.

The main idea is that there are a total of $m + n$ hosts available to perform the necessary computations during the session, m of which were designated by the service provider to handle all requests from all service users, and n of which where allocated by the service user at the local site for that session. The motivation for choosing this particular model is that it gives the service user the option to contribute resources that might help reduce the response times for their service requests. In various cases, a service user who needs better response times can get it by increasing the value of n.

Note that our model is a simple generalization of some other grid service models. For example, the usual client/server Internet model assumes $m = n = 1$, and Netsolve, a network-enabled solver toolkit from University of Tennessee [6] has $m \geq 1$ and $n = 1$.

3 Active frames

Active frames provide a simple mechanism to send work and data to compute hosts. With this flexible mechanism the services can take advantage of the resources in hosts at the local and remote sites. An active frame is an application-level transfer unit that contains program and associated application data. Section 3.1 describes the active frame interface. Active frames are processed by compute servers called *frame servers*. The frame servers are user-level processes that run on compute hosts. Figure 2 illustrates the structure of a frame server. A server consists of two components: an application-independent interpreter that executes active frame programs, and an optional collection of application-specific routines that can be called from the frame program. Section 3.2 describes the details of theframe server implementation. The active frame mechanism can be extended to build sophisticated services through the use of specialized active frames and application libraries that are loaded into the servers. Section 3.3 describes this extension mechanism.

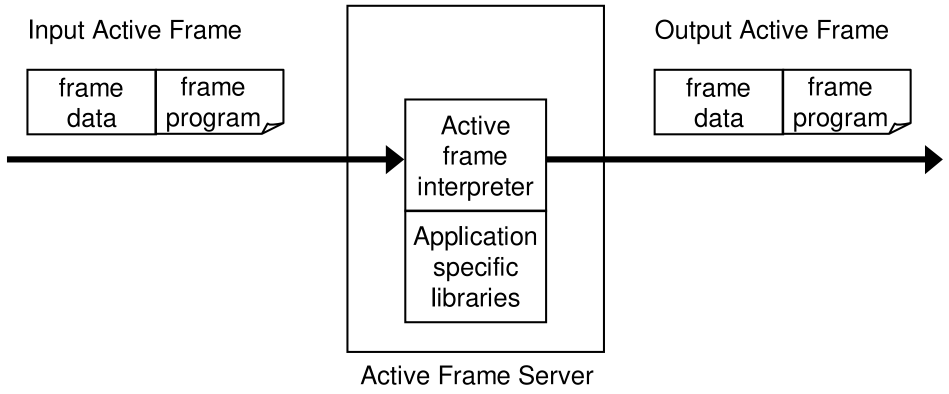

Figure 2: Active frame server.

3.1 Active frame interface

The active frame interface declares a single `execute` method.

```
interface ActiveFrame {
  HostAddress execute(ServerState state);
}
```

The specific frame's execute method is called when the frame arrives at the server. The execute method is the entry point for frame execution and can be used to process the frame's data. This method receives one parameter of type `ServerState`. The state object provides a soft storage service that frames can use to keep arbitrary data. The soft storage mechanism is described in more detail in section 3.2.2. When the frame execution finishes, the server forwards the frame to the server specified by the address returned by the execute method.

This simple mechanism allows applications to specify what operations to execute on the data and where to execute them. Heavyweight services can use this mechanism to move computation along with the data to servers, taking advantage of the the extra resources these compute servers offer.

Application-specific frames provide the appropriate implementation of the execute method. For instance, long-lived frames can be used to install services in the compute hosts that satisfy requests from other clients. Other compute-intensive services like future search engines and data mining could be implemented with more short-lived frames that crawl the databases searching for specific information. Visual animation of physical processes could be implemented with a stream of multiple active frames where each frame processes part of the dataset.

3.2 Frame servers

Frame servers are processes that receive, execute and forward active frames. We implemented the frame servers as user-level processes using Java [18]. This has allowed us to easily implement the active frame mechanism while avoiding OS and machine specific dependencies. Most of the functionality of the active frame mechanism is facilitated or built directly on top of Java's features. For example, the marshalling of `void FrameServer.send(ActiveFrame frame):`

Execute and send an active frame to the server returned by its execute method.

`Object ServerState.get(String id):`

Retrieve an object from the server soft storage.

`void ServerState.put(String id):`

Add an object to the server soft storage.

Figure 3: Frame server API

active frames is done using Java's serialization protocol and the frame's code distribution mechanism gets leverage from Java's dynamic byte-code loading. Frame servers provide the following functionality: (1) frame processing and forwarding, and (2) soft storage.

3.2.1 Frame processing and forwarding

An application starts a new server in the same process by creating a new instance of a `FrameServer` object. The frame server processes outgoing frames coming from the application and incoming frames arriving from the network. The application calls the tt send method of the frame server (See Figure 3) to initiate the process of a frame. Here is how an application might create and start a frame server:

```
1: FrameServer server = new FrameServer(3000);
2: ActiveFrame frame  = new SampleFrame();
3: server.send(frame);
```

The application starts a server on the local host using port 3000 (line 1), and then creates and sends a new active frame (lines 2,3). The server's `send` method invokes the frame's `execute` method before sending the frame. Then the server sends the frame to the address returned by the execute method if the address is not null.

Frames are transfered among servers using TCP/IP. At run-time, a frame server waits on a well-known port for an active frame to arrive from the network. When a new frame arrives, the server reads the program and data contained in the frame; then the server interprets and executes the frame program. We built the frame servers using the standard approach where each frames is executed in a separate thread of control. This process is described next. Figure 4 shows an active frame in the first server to the left that is about

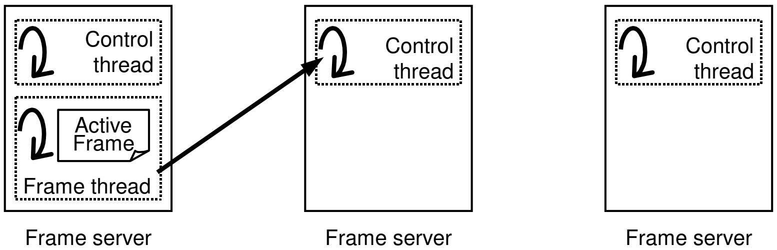

Figure 4: Frame transmission initialization

to be sent to the server in the middle. The *control thread* in the receiving server (middle) listens for new connections. The sending server establishes a TCP connection to the receiving server to transfer the frame.

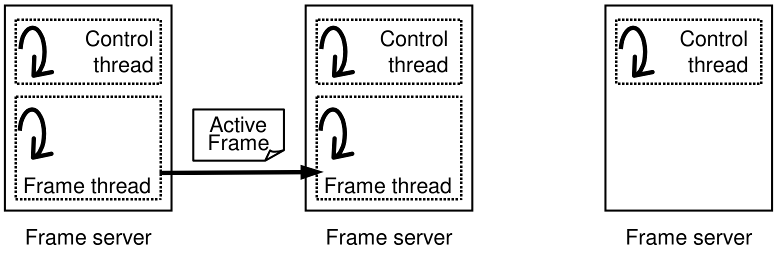

Figure 5: Frame transmission

In Figure 5 the control thread in the receiving server creates a new thread, labeled *frame thread* in the figure. The control thread hands the connection to the frame thread and returns to wait for new connections. The frame thread unmarshalls the frame from the stream using a `FrameInputStream` object. The `FrameInputStream` object performs two functions: (1) It takes care of the details of unmarshalling the frame, and (2) it dynamically loads the code needed to execute the frame.

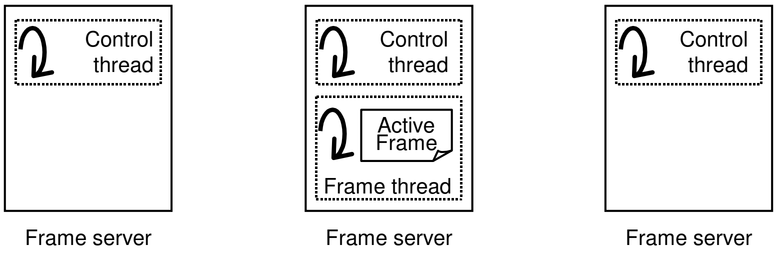

Figure 6: Frame execution

Once the frame is unmarshalled, the frame's `execute` method is called in the frame thread (Figure 6). The execute method performs the desired computation (i.e., transform the data).

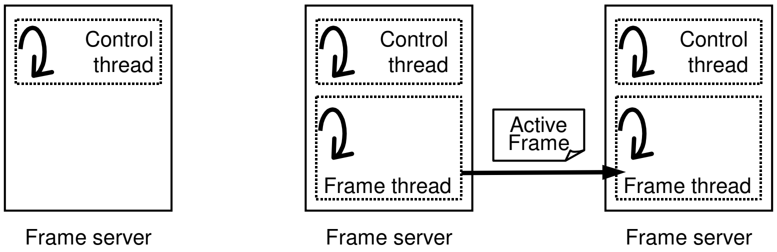

Figure 7: Frame forwarding

Once the execute method finishes it returns the address of a frame server. If this address is not null the server forwards the frame to this address. The frame thread uses the address to establish a connection to the next server and then creates a `FrameOutputStream` object to marshall the frame into the stream (Figure 7).

3.2.2 Soft storage

The soft storage is a mechanism to save arbitrary data temporarily at the frame servers. Applications can use this mechanism to keep state at the servers. The storage accepts key/value pairs to store the data. Later the same key can be used to retrieve and modify the data or remove it from the storage. An object of type

`ServerState` is passed to the frame's `execute` method. The frame uses methods in the `ServerState` class to `put`, `get` and `remove` data from the store (See figure 3).

The soft storage, as implied by its name, makes no guarantees about preserving the data it keeps. This implies that application frames should be prepared to recover information that has been flushed from the storage. The entries in the soft storage are *lease based* [7], meaning that the entries are kept only for a finite period of time as defined by the lease. At the end of the specified period the entry is removed if its lease has not been renewed. Although the store makes an effort to keep an entry until the end of the lease period, the entry could be removed before the lease expires.

This simple mechanism provided by the soft storage is useful for sharing information between different frames that belong to a session or a type of application. For example, frames belonging to a data stream can store and retrieve parameters and performance metrics related to the processing of the stream. Here is a sample frame that uses soft storage to keep state between execution of frames:

```
1:class SampleFrame implements ActiveFrame {
2:  HostAddress execute(ServerState state) {
3:    Integer estimate = (Integer)state.get(ESTIMATE_ID);
       ... // set application parameters
4:    Integer execTime = compute();
5:    Integer newEstimate = computeEstimate(estimate, execTime);
6:    state.put(ESTIMATE_ID, newEstimate)
      ...
    }
  }
```

In this example, the frame uses the `get` method (line 3) to obtain a previously stored estimate of the time it will take to complete a computation. After the frame finishes the computation (line 4) it updates and stores the estimate in the server's soft storage using the `put` method (lines 5-6).

3.3 Extension support for active frames

In order to create higher-level services using active frames, the frames need to perform application specific operation at the servers. This is a crucial feature because it allows the use of existing packages and integration with other systems. For example, visualization algorithms are complex and difficult to develop, a visualization service can incorporate existing visualization packages such as vtk [29], AVS [32] and openDX [1].

Active frames use two mechanisms to extend the basic functionality of the servers: dynamic code loading and *application libraries*. The implementation of these two mechanisms relies on features that are central to Java: dynamic class loading, the object serialization protocol and JNI (Java Native Interface) [21, 17].

3.3.1 Dynamic code loading

The code needed to execute an active frame program is loaded on demand and cached by the frame servers. The program carried by the active frames is meant to be a glue code that calls into the application specific libraries to implement a service or a distributed application. In general, active frame programs are expected to be relatively small and can be easily loaded over a network. The code loading mechanism used with active frames is similar to the one used by Java's RMI (Remote Method Invocation) [12]. This mechanism is supported by the JVM's (Java Virtual Machine) [22] ability to dynamically resolve and link classes at run time.

[1] See http://www.research.ibm.com/dx

The servers load the code for the frames using a signaling protocol separate from the one used to forward the frames. However the code loading mechanism is tightly coupled with the marshalling protocol used to forward the frames. The frames are marshalled in the data using Java's object serialization protocol. This protocol has the option of including a *class annotation* along with the object marshalled in the data stream. This annotation is used to indicate where to find the code for a particular frame. With the use of a separate signaling protocol the class belonging to a frame can be cached for future use avoiding the cost incurred in fetching, loading and interpreting the class the first time. The processing of a frame is slowed down considerably the first time a particular class is loaded, especially if the classes have to be transfered from a remote class server. This can be solved by prefetching the classes ahead of time so by the time a frame arrives the the server the corresponding classes have already been loaded.

When a frame server forwards an active frame to another server, the sending server uses a `FrameOutputStream` to marshall the frame into a byte stream using Java's serialization protocol. The serialization protocol is implemented in `FrameOutputStream`'s superclass, `ObjectOutputStream`. `FrameOutputStream` overrides the `annotateClass` method. This method is called once for each unique class in the stream the first time an object of that class is marshalled. The method adds a set of URLs to the stream. These URLs are commonly known as *codebase*. The codebase can be used by the unmarshalling process to locate the appropriate code for the frame or other objects included with the frame. The `FrameOutputStream` obtains the URLs by querying the class loader of the class to annotate; if the class loader cannot provide a codebase, then a default value is used.

The receiving frame server uses a `FrameInputStream` to unmarshall the frame from the incoming stream. The `readObject` in the superclass `ObjectInputStream` implements the unmarshalling protocol. The `readObject` uses information included in the data stream to determine the class of an object that it is reading. When a new class is found in the stream the `resolveClass` method is called to load the corresponding class. The `FrameInputStream` class overrides the `resolveClass` method, which is the counterpart of `FrameOutputStream's annotateClass` method. The `resolveClass` method reads the codebase from the byte stream and then it delegates the class resolution and loading to a `FrameClassLoader`.

`FrameClassLoader` is a subclass of `java.lang.ClassLoader` that loads classes from a specified set of URLs (codebase). In JDK (Java Development Kit) version 1.2 [8] a new class, `java.net.URLClassLoader`, was introduced which provides this functionality. The loaders cache classes that have already been fetched to avoid the expensive process of contacting the class server and dynamically validating and linking the code. If the class cannot be found an exception is raised.

The protocol used to transfer the class files from their repository is independent of the class loader; this means that the classes can be transfered over HTTP, FTP or other protocol as long as the name can be specified as a URL and there exists an installed handler for the protocol. In our setup the classes are stored at a central repository and are made available to the frame servers through a simple HTTP server.

3.3.2 Application libraries

The dynamic code loading mechanism explained in the previous section is well suited for small programs, like glue code, carried in the active frames. However, in order to implement useful services, the frames need to perform complex application-specific operations. For example a visualization application needs a series of filters to process a dataset and produce a 3D model or an image out of the data.

The frame servers can be extended to provide a richer set of operations to Active Frames. The frame servers are extended by adding application specific routines in the form of libraries that are colocated with the servers. The servers load the libraries at initialization time. This mechanism makes it possible (1) to use of existing code and libraries; (2) take advantage of platform and application specific optimizations that can be implemented in the libraries; and (3) avoid the overhead involved with the flexibility of the runtime environment, namely interpretation and compilation. These approach is well-suited for compute intensive tasks like remote visualization. In many cases the size of the library makes it impractical to load the code on demand over a slow network. Thus extending the servers with libraries provides a richer API that application-specific active frames can use without incurring a high overhead.

Active Frames can call methods in the libraries through JNI, which is Java's mechanism to execute platform dependent code, also called native code. To access routines in existing application libraries a thin layer of wrapper code translates Java method calls into library calls.

The application libraries have the following three components: (1) the Java wrappers; (2) the C method wrappers; and (3) the application routines. The *Java wrappers* declare the interface for the routines in the libraries. These wrappers do not provide any implementation of the routines, they just declare that the routines exist and that they are part of the library. The *C method wrappers* are a series of stub routines with naming and parameter conventions used by JNI. These wrappers are responsible for conversions between Java types and native types. The *application routines* provide the actual implementation of the desired function. If the application libraries are written in an object oriented language like C++ then a direct mapping from exported library classes to Java classes is the natural approach.

Libraries are loaded into the frame servers at startup time using the `System.load` method of the Java API. An active frame accesses a routine in the application library by calling the corresponding method in a Java wrapper object. The Java runtime calls the appropriate method wrapper with the specified parameters. Finally the method wrapper calls the specific function in the library. When the library routine returns, the wrapper passes the return value back to the Java runtime. Here is the basic form of a Java wrapper:

```
1:class SampleWrapper {
2: private native void libMethod0(int parameter);
3: public void libMethod(int parameter) {
     // Check parameters and then call the native method.
4:   return this.libMethod0(parameter);
   }
  }
```

The Java wrappers are objects that provide access to the library routines through instance methods that are declared `native`. In the example above the class `SampleWrapper` declares `libMethod0` to be native (line 2). Objects from other classes can access the native method through the public method `libMethod` (lines 3-4). The method `libMethod` can be used to check the parameters if necessary.

```
1:extern "C" JNIEXPORT void JNICALL
2:Java_SampleWrapper_libMethod_10(JNIEnv *env, jobject obj, jint par) {
    // call the actual library method
3:  libMethod( par );
  }
```

The method wrappers are small functions that conform to the naming and parameter passing convention of JNI. These functions are called whenever a Java program calls a method declared native in the corresponding wrapper objects. A method wrapper uses the parameters it receives from the JVM to construct the parameters needed to call the corresponding application routine in the library. After the library routine returns, the method wrapper converts the return value to conform with the JNI convention. The method wrappers are written in C/C++, compiled into platform specific instructions and linked with the application library to generate a dynamic library. The example above shows the corresponding method wrapper for the native method of the sample class. The function declaration follows the JNI convention (lines 1-2). The application routines are at the core of the library and carry out the actual computation (line 3).

4 Remote visualization service

In many situations the amount of data collected by a particular process is extremely large. The size of the data makes it difficult for a human being to interpret and analyze. Visualization techniques are very useful to analyze large amounts of data because they highlight interesting features of the data. The goal of a remote visualization service is to enable users to create in their desktops visual representations of massive datasets that are stored remotely.

In general, the ability to interact with the visualization process increases the user's understanding of the dataset. For example, changing parameters in the visualization or exploring the dataset from a different viewpoint might reveal features of the dataset that were hidden. In order to allow this kind of interaction, the delay between the user input and the output has to be within reasonable bounds.

Users would like to analyze their datasets interactively from their personal desktop and laptop computers using the computational resources they have access to. The visualization process performs a sequence of compute intensive transformations to the data to produce an image. The size of the datasets and the computational requirements of the visualization programs difficult the creation of interactive applications that run on a typical desktop workstation. The bottom line is that very often a user is far away from datasets he wants to access. Often the datasets are difficult to transfer over the network to the users site, so they have to remain at a remote site where the data was collected or generated.

The process of interactive visualization can be thought of as series of queries to a massive dataset. Before the data is presented to the user it must be processed by a sequence of filters. Remote visualization of scientific datasets is a heavyweight interactive service. We have created a remote visualization service that we call Dv (Distributed visualization) [1]. Dv enables the user to visualize the massive scientific datasets that are stored at remote sites. The service is provided under the resource model described in section 2, thus the visualization process is partitioned among the hosts at the remote and local sites to satisfy the resource requirements.

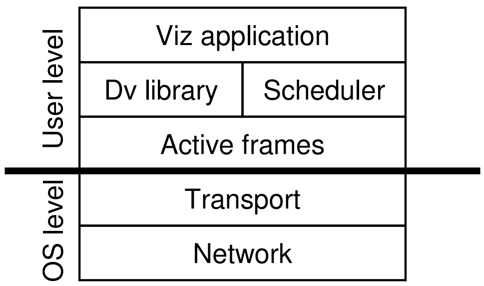

Figure 8: Dv service composition

Dv operates at the application level using existing network services, thus it does not require changes in the current network infrastructure. Figure 8 shows the composition of the Dv service.

- The *visualization application* specifies how process the set of filters required to transform the data into images.
- The *Dv library* is a collection of routines to process and transfer datasets. In order to process the data Dv uses vtk, a powerful and comprehensive open-source visualization library [29]. Section 6 describes the application model defined in the library.

- The *Scheduling interface*, described in section 7, allows the use of application-level information to select the resources required to execute the visualization.
- Dv aggregates resources from multiple hosts to execute the visualization process. Dv uses *active frames* to ship data and computation to the compute hosts.

5 Motivating application

Scientists and engineers frequently use computer simulations to solve complex models for their problem of interest. The output of these simulations result in large amounts of data. For example, the Quake project at CMU [4] uses supercomputers to simulate the motion of the ground during strong earthquakes. Quake simulations produce datasets on the order of hundreds of gigabytes to terabytes of floating point numbers that represent the displacements of points in the earth during an earthquake. Visualization, data mining, statistical analysis and other methods must be used to obtain insight of the data.

Quake simulations manipulate large unstructured finite element models. For a moderately sized basin such as the San Fernando Valley of Southern California, the models consist of 15M nodes and 80M tetrahedral elements. A simulation of 60 seconds of shaking can generate on the order of 6 TB of data. The output dataset consists of a large 3D unstructured tetrahedral mesh that models the ground and a series of *data frames*. Each frame records the state of the mesh at a given time-step in the simulation. A frame contains data values associated with each point in the mesh. These values describe features of the points in the mesh, such as the displacement amplitude (i.e., the amount of motion) at a point in the earth. The series of frames describes the state of the mesh over time.

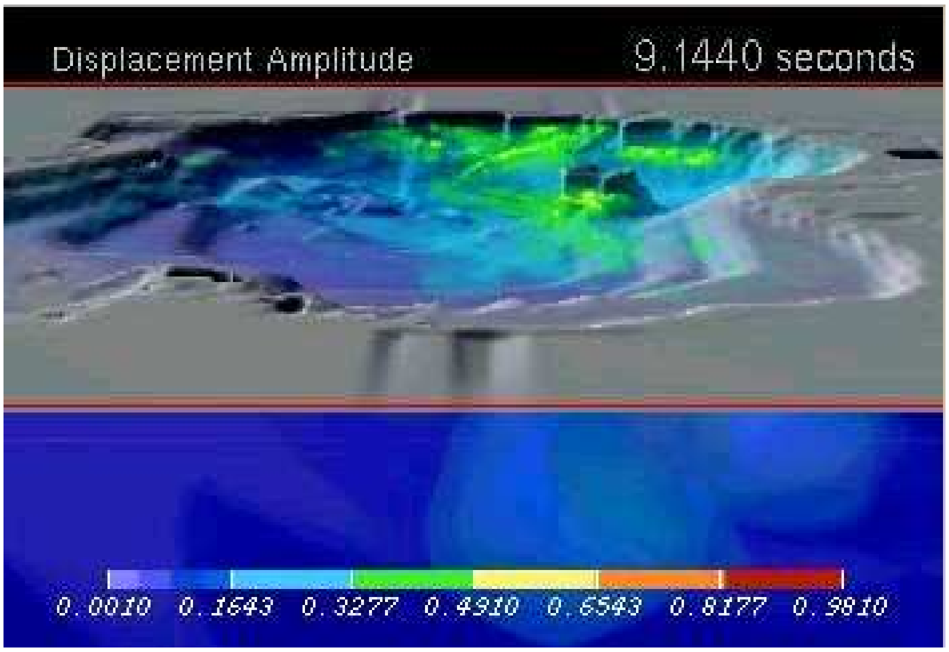

Figure 9: Quake visualization snapshot

Figure 9 shows a visualization of the displacement amplitude of the ground during the 1994 Northridge aftershock in San Fernando Valley. The top part of the picture shows the density of the terrain near the surface. Underneath different tones of gray represent the displacement of ground's magnitude as the wave propagates to the surface.

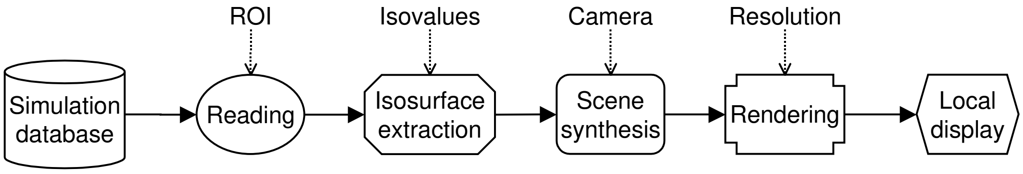

Figure 10: Flowgraph for a Quake visualization

In order to produce an image, the visualization process applies a series of transformations to the data. These transformations are usually expressed in the form of a *visualization flowgraph*. Figure 10 shows the *flowgraph* of the transformations involved in a simplified visualization of a Quake dataset. The ground motion values are stored in the remote dataset. A user at the *local site* interactively request the visualization of a ROI (*region of interest*) in the dataset. The region of interest is expressed both in space (volume/plane of the ground) and in time (set of frames). If the user requests data from a single frame, then the result is a still image. If the user requests multiple frames then the result is an animation. The first stage of the flowgraph reads a subset of the dataset as indicated by the ROI. The next stage computes isosurfaces based on soil densities and ground displacement amplitudes to highlight characteristics in the data. The next stage in the flowgraph synthesizes the scene according to various scene parameters such as point of view and camera distance. Finally, the scene is rendered from polygons into pixels that are displayed on a monitor. The resolution of the displayed image is determined by a resolution parameter.

Figure 11 shows a simplified version of the C++ code used to implement the flowgraph shown in Figure 10. The code shown here uses vtk filters to produce the visualization.

In lines 2-7 the vtk objects required to visualize the dataset are created. `dvQVolumeReader` (line 2) is the object that reads the quake dataset. The `vtkContourFilter` object (line 3) performs the isosurface extraction on the dataset. The `vtkPolyDataMapper`, `vtkActor` and `vtkRenderer` objects (lines 4-6) work together to synthesize and render the 3D scene. Finally, the `vtkRenderWindow` object (line 7) displays the resulting scene. In lines 8-24 the filters are setup with the initial parameters. Also the objects that form the flowgraph are connected together. Line 8 specifies to the reader object the name of the dataset to visualize. The contour filter generates as many isovalues as specified by the `_numIsos` parameter (Line 10). These values values are evenly distributed between the given minimum and maximum values (`_minval` and `_maxval` parameters). For the sake of simplicity the scene parameters have been omitted. If no light and camera position are specified the renderer creates a default light and a camera to render the scene. Also the default values for the window size and resolution parameters are used to display the final image.

In line 9, the reader and the contour filter are connected by assigning the output of the reader to the input of the contour filter. Similarly, in line 11 the contour filter is connected to the data mapper. In line 12, the actor is associated with the mapper and in line 13 the actor is added to the list of actors in the renderer. Finally in line 14, the renderer is associated with the display window. The function `displayTimeStep` (line 15) is called when the user specifies a data frame corresponding to a particular time step in the simulation. This parameter is updated in the reader (line 16) and the `Render` method in the window object (line 17) triggers the execution of the flowgraph.

Figure 12 shows a Dv version of the ground motion visualization generated by the program in Figure 11. In this example, the Dv client sends an active frame with a request to the remote server where the dataset is located. This active frame generates one or more active frames as a response. Each response frame carries a program that assigns the reading component of the visualization flowgraph to the first remote Dv server, the isosurface extraction stages to the second Dv server, the scene synthesis, the rendering and the display stages to the local Dv client.

```
void setup()
  {
   dvQVolumeReader*   reader        = dvQVolumeReader::New();
   vtkContourFilter*  contourFilter = vtkContourFilter::New();
   vtkPolyDataMapper* polyMapper    = vtkPolyDataMapper::New();
   vtkActor*          actor         = vtkActor::New();
   vtkRenderer*       renderer      = vtkRenderer::New();
   vtkRenderWindow*   renWin        = vtkRenderWindow::New();

   // set up the filters
   reader->SetFileName( _datasetname );
   contourFilter->SetInput( reader->GetOutput() );
   contourFilter->GenerateValues( _numIsos, _minval, _maxval );
   polyMapper->SetInput( contourFilter->GetOutput() );
   actor->SetMapper( polyMapper );
   renderer->AddActor( actor );
   renWin->AddRenderer( renderer );
  }

void displayTimeStep(int timestep)
  {
   reader->SetTimeStep(timestep);
   renWin->Render();
  }
```

Figure 11: Sequential quakeviz program

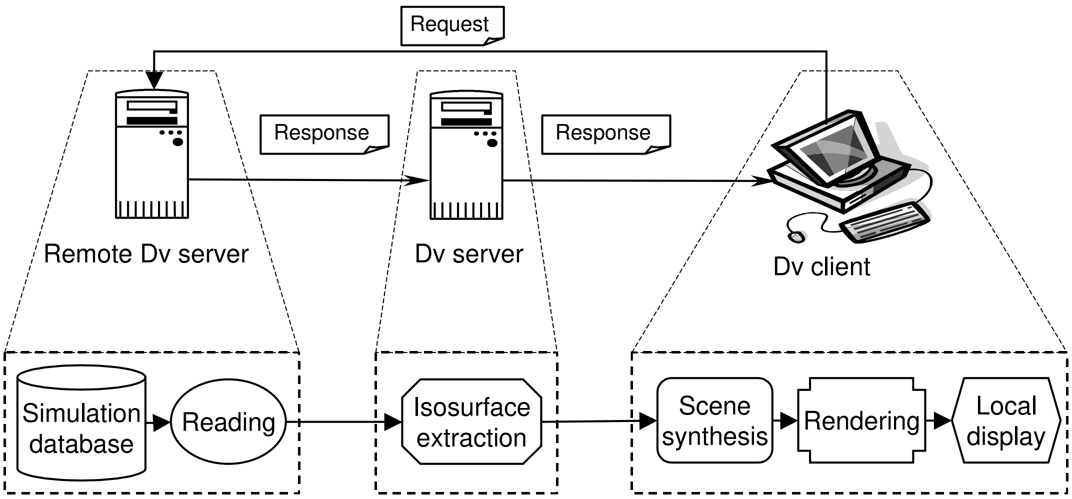

Figure 12: Example Dv Quake simulation.

Method
`void DvFrameServer.sendData(vtkDataSet data, DvHostAddress address, int sessionId, int dataId)`
`vtkDataSet DvFrameServer.getDataSet(int sessionId, int dataId)`
`HostAddress DvFrameServer.getLocalAddress()`

Figure 13: Dv server API

6 Dv library

The Dv library contains the routines to build the visualization service. The service is provided on a collection of compute hosts running *Dv servers*. A Dv server is an active frame server extended with the Dv library. Dv servers are explained in section 6.1. The user interacts with the visualization process through the Dv client. The Dv client is an active frame server extended with an interface to display the visualization. Section 6.2 describes the Dv client. In order to provide a particular visualization service with Dv, the application designer has to transform de application and divide it in various components as defined by the Dv library. The visualization components and application model are described in section 6.3. To initiate a Dv session, the Dv client sends a special kind of active frame, called *request handler*, to a Dv server that is colocated with the remote dataset (the *dataset server*). During the course of a visualization session, the Dv client sends a series of *request frames* to that Dv server. Each request frame contains visualization parameters such as region of interest and resolution, a description of the visualization flowgraph, and a scheduler that assigns flowgraph nodes to Dv servers. The request handler produces a sequence of one or more *response frames* depending on whether the client requests an animation or a still image.

6.1 Dv servers

A Dv server is a specialized frame server that has been extended to provide additional services for visualization applications. A Dv server is extended with visualization libraries using the mechanism described in section 3.3.2. In our current implementation the Dv servers are extended with vtk libraries. The servers are also extended with routines to transfer vtk datasets between servers.

Using existing visualization libraries has various advantages. We do not have to rewrite complex visualization routines to support remote visualization services. Existing libraries are more reliable since they have been widely used by the visualization community. These libraries have been optimized over their life cycle to be fast and memory efficient, which results in an overall performance gain that would be difficult to achieve if we wrote our own visualization routines.

The visualization libraries define their own data types and provide the routines to operate on those data types. In vtk, the data types are all sub-types of the `vtkDataSet` class. Since datasets need to be moved between servers, we created a series of marshaller objects to transfer the datasets over TCP streams. There is one marshaller / demarshaller pair for each vtk dataset type. These marshallers are C++ objects that are included in the application libraries used to extend the Dv servers. The Dv server's API provides a simplified interface to transfer the datasets between servers using these marshallers.

At the sender side the application can use the `sendData` method to initiate a dataset transfer to a particular host (See Figure 13). This method takes four parameters. The first parameter is a reference to the dataset to send, the second parameter specifies the destination host, the third and fourth parameter are used to identify and retrieve the dataset at the receiving server. Each dataset is associated with a session and a frame it belongs to.

At the receiver's side a `DataFrameReceiver` object reads the dataset using the corresponding de-marshallers for the data type. The dataset is buffered for later retrieval. Later an application frame can use the `getDataSet` method (See Figure 13) to retrieve the dataset. If an application's frame calls the `getDataSet` method before the dataset has been received, the calling frame waits for a predetermined period of time for the dataset to arrive. An application can also use the extended API (not shown in table) to specify the timeout value it is willing to wait for the dataset.

An application frame can obtain the address of the Dv server it is executing on through the `getLocalAddress` method. This method returns a `DvHostAddress` object, which is a subclass of `HostAddress`.

6.2 Dv Client

The Dv client displays the output of the visualization and allows the user to interact with the visualization process. The Dv client is in essence a frame server with the following functionality: (1) user input processing, (2) visualization output rendering, (3) session management, and (4) data frame buffering and reordering.

The usert interface of the Dv client can be easily customized. We expect the Dv client to be useful to create basic user interfaces for simple visualization services. More complex interfaces and interaction might require the construction of an application-specific client from the scratch. A visualization designer can create the user interface appropriate to visualize a particular dataset. If no user interface is provided, the client creates a default window to display all the data frames that arrive. The interaction with the visualization process using the default window is limited to rotate, pan and zoom into the dataset, however the user cannot modify any of the visualization parameters directly.

The Dv client associates frames with a session. The client starts a session by sending a `RequestHandler` frame to a server. Usually this server is colocated with the dataset to be visualized. The request handler is a long-lived frame that accepts requests from the client and sends response frames back to the client. The response frames execute on the compute hosts along the path back to the client. The response frames also execute when they arrive to the client, the same way they do in any other frame server. The request handler, request and response frames are described in detail in section 6.4

The Dv client provides a call back mechanism to announce the arrival of response frames. The application designer provides the implementation for the `FrameListener` interface shown below.

```
public interface FrameListener {
  void frameArrived(ResponseFrame frame, vtkDataSet data, int id);
}
```

The `frameArrived` method of the registered frame listener is called after the frame's execute method returns. The first parameter for this method is the frame that just arrived, the second parameter is the output dataset produced by the response frame, the last parameter is the identifier of the dataset. Using this mechanism the application designer can customize the behavior of the client. For example, one particular implementation of this interface could render the dataset once it arrives to the client. Another implementation could update a field in the screen with the current frame number.

The `ClientBuffer` is a helper class that implements the `FrameListener` interfaces. The client buffer orders the response frames received from the server. This class is particularly useful to create animations, where the response to a request is a stream of frames that have to be displayed in order at a specific time.

6.3 Visualization components

In order to build a Dv application, the application designer needs to implement the following components: (1) a data source, (2) a visualization flowgraph, (3) requests, and optionally (4) a scheduler and (5) a customized a graphical user interface. These components are present in the toolkit in the form of interfaces, leaving the concrete implementation to the application specific needs.

6.3.1 Data source

The data source reads the dataset. The interface for the data source is shown below:

```
public interface DataSource extends java.io.Serializable {
  public vtkDataSet getDataFrame();
}
```

The `DataSource` interface extends the `Serializable` interface so data sources can be marshalled and transmitted along with the active frames. This interface declares a single `getDataFrame` method that returns a reference to a `vtkDataSet` object. A particular implementation of this interface could read the data from a database, a file or directly from output of a simulation process. The data source's methods are executed in a frame server that has access to the data. Usually the data source object exists throughout the duration of a session and this object keeps state like open files, established database connections, current loaded frame, etc.

```
 1: public class QuakeSource implements dv.DataSource {
 2:  String datasetName = null;
 3:  int    timeStep    = 0;

 4:  public vtkDataSet getDataFrame() {
 5:    if (this.reader==null) {
 6:       reader = new dvQVolumeReader();
 7:       reader.SetFileName(this.datasetName);
       }
 8:    this.reader.SetTimeStep(this.timeStep);
 9:    this.reader.Update();
10:    return this.reader.GetOutput();
     }

11:   public void setTimeStep(int step) {
12:    this.timeStep = step;
     }
   }
```

In the example above, the `QuakeSource` class is used to read the output files generated by Quake simulations. The simulation's output files contain an unstructured tetrahedral mesh and a series of scalar values associated to the vertices of the mesh for each time step of the simulation process. The `QuakeSource` class implements the `getDataFrame` method (line 4) declared in the `DataSource` interface. In line 5 this method checks if an instance of a `dvQVolumeReader` has been created. If no reader has been created one is instantiated and initialized (lines 6 & 7). `dvQVolumeReader` is a class in the application library that reads the quake files and produces vtk datasets as output. In line 8 the desired time step to read is specified to the reader. The reader's `Update` method (line 9) loads the actual dataset. Finally and the `GetOutput` returns a reference to the dataset. The `QuakeSource` class also provides methods to specify the time step of the simulation of interest to the user.

6.3.2 Visualization flowgraph

The visualization flowgraph describes the set of transformations to perform to the datasets. The flowgraph contains the actual program, in the form of dataset filters, to create a visual representation of the datasets. The visualization designer partitions the flowgraph into nodes, each containing one or more filters that are applied to an input dataset to produce an output dataset. Nodes in the flowgraph are independent of each other and connected only by the input and outputs datasets, allowing the nodes to be executed separately. For instance the first node of the flowgraph could be executed on host A, when execution would finish the output would be transfered to host B where the second node would execute using the first node's output as input for the second node. Also, multiple consecutive nodes in the flowgraph can be executed in the same host.

This model offers great flexibility allowing the flowgraph to be scheduled in many different ways according to resource availability. However, the flexibility of the model has a cost. Not all applications fit naturally in this programming model and the model makes it more difficult to reason about the control flow of the program. We are still evaluating this model to improve its ease of use or adopt a different model if necessary.

The `Flowgraph` interface declares two methods: `buildNode` and `getNumberOfNodes`.

```
public interface Flowgraph extends java.io.Serializable {
  public int getNumberOfNodes();
  public vtkDataSet buildNode(int nodenumber, vtkDataSet input)
    throws FrameException;
}
```

The `getNumberOfNodes` returns the number of nodes in the flowgraph. This method is used primarily by the schedulers. The `buildNode` method instantiates a single node of the flowgraph. The first parameter of this method is a zero-based index that identifies the node to be instantiated. The second parameter is a handle to the input for the node to build. The method returns a handle to the output of the node. Note that at this point the flowgraph's node has only been constructed and it has not been executed yet, thus the return output dataset is still empty.

Here is an example of how a flowgraph can be implemented.

```
public class SampleFlowgraph implements Flowgraph {
  public int getNumberOfNodes() {
    return 2;
  }

  vtkDataSet buildNode(int nodeNumber, vtkDataSet input) {
    switch (nodeNumber) {
      case 0: return buildNode0(input);
      case 1: return buildNode1(input);
      default: throw new FrameException(``Invalid node number'');
    }
  }
```

```
     /* Build node 0 to extract the region of interest.
      * Interesect the input with a plane (ROI). */
 9:   vtkPolyData buildNode0(vtkDataSet data) {
10:     vtkCutter cutter = new vtkCutter();
11:     vtkPlane plane   = new vtk.vtkPlane();
12:     plane.SetOrigin( 0.0, 0.0, this.zvalue );
13:     plane.SetNormal( 0.0, 0.0, 1.0 );
14:     planeCut.SetInput( data );
15:     planeCut.SetCutFunction( plane );
16:     return planeCut.GetOutput();
      }

      /** Build node 1: Extract isosurfaces */
17:   vtkPolyData buildNode1(vtkDataSet data) {
18:     vtkContourFilter filter = new vtkContourFilter();
19:     filter.SetInput( data );
20:     filter.GenerateValues( this.isovalues, this.low, this.high );
21:     return filter.GetOutput();
      }
    }
```

The sample flowgraph shown above has two nodes. The first node extracts a region of interest in the dataset. The second node decimates the dataset. The `buildNode` method uses the node number parameter to determine the node to build (lines 4-8). The flowgraph can be implemented using visualization filters and routines in the application library. For example, in the `buildNode0` method a `vtkCutter` node is instantiated from the vtk libraries (line 12). The mechanism to connect the different nodes of the flowgraph is explained in section 6.4.3.

6.3.3 Request

A request specifies the parameters of the visualization. Requests might be used to change the ROI, isovalues and other parameters in the visualization flowgraph. Requests can also be used to change parameters in the data source or in the scheduler. Here is a simplified version of the `Request` class.

```
public class Request implements java.io.Serializable {

  protected DataSource source;
  protected Flowgraph flowgraph;
  protected Scheduler scheduler;
  protected boolean override;
  protected int sessionId;
  protected int frameId;

  /** Gets a new or updates an existing flowgraph. */
  public Flowgraph getFlowgraph(Flowgraph current) { ... }

  /** Gets a new or updates an existing data source. */
```

```
 9:  public DataSource getSource(DataSource current) { ... }

     /** Gets a new or updates an existing scheduler. */
10:  public Scheduler getScheduler(Scheduler existing) { ... }

     /** Does this request require more response frames */
11:  public boolean hasMoreFrames() { ... }

     /** Is this an overriding request? */
12:  public boolean overrideRequest() {
13:    return this.override;
     }
   }
```

The `Request` class implements the `Serializable` interface (line 1) allowing it to be marshalled / unmarshalled using Java's serialization protocol. The request carries with it a data source, a flowgraph and a scheduler (lines 2-4). The corresponding `get` methods (lines 8-10) are used to retrieve these fields from the request. These `get` methods may change parameters to the field. For example the `getFlowgraph` method receives the current flowgraph, if any, as a parameter. These method can change parameters to the existing flowgraph, or replace the existing flowgraph with the one carried by the request. The default implementation of these methods return the object passed as parameter without any modification if it is not null, otherwise the corresponding object carried in the request is returned. The request also has fields to identify the session and the response frames (lines 6-7). Since a request can generate multiple responses, the `hasMoreFrames` method (line 11) is used to determine if more response frames are to be sent back to the client for this request. Whether a request may override outstanding requests is determined calling the `overrideRequest` method (lines 12-13).

6.3.4 Scheduler

The scheduler maps the execution of flowgraph nodes to compute hosts. Flowgraph nodes can be executed independently. These allows to execute a flowgraph node in one host and then execute the next node in a different host. Also, multiple nodes of the flowgraph can be executed in the same host. The response frames (described in section 6.4.3) use the scheduler to determine where each flowgraph node is executed. Here is the interface of the Scheduler:

```
public interface Scheduler {
  DvHostAddress getHost(int hopCount)
}
```

The scheduler declares a single `getHost` method. This method receives a hop count parameter that indicates the number of hosts the corresponding frame has traveled so far. A value of zero for the node number indicates that the frame is at the *source server*. The `getHost` method returns the address of the next host where to frame is to be sent and executed. Frame scheduling is described in more detail in section 7.

6.4 Dv frames

Dv defines three types of frames for the execution and control of the visualization application: *request handler frame*, *request frame* and *response frame*.

6.4.1 Request handler frame

To initiate an application session, the client sends a request handler frame to a frame server that has access to the dataset to visualize. The request handler is a long-lived active frame that satisfies the client requests for the duration of the session.

request handler data	initial request	data source	visualization flowgraph	scheduler

Figure 14: Request handler frame

Figure 14 shows the structure of a `RequestHandler` frame. The handler frame may carry a request with itself. This first request specifies the initial data source, visualization flowgraph and scheduler. The request handler initializes and maintains state for the session on behalf of the client. For example, the request keeps a reference to the data source, which maintains open file handles or database connections.

```
public class RequestHandler implements ActiveFrame {
  private Request firstRequest;  // The first request
  private transient RequestQueue requestQueue;

  public HostAddress execute(ServerState state)
    throws FrameException
  {
    // Register the handler in the server state
    this.requestQueue = new RequestQueue();
    this.requestQueue.addRequest(this.firstRequest);
    state.addObject(this.sessionId, this);

    FrameServer server = state.getServer();

    int frameID;
    Flowgraph flowgraph = null;
    DataSource source   = null;
    Scheduler scheduler = null;
    Request request     = null;

    while((request = this.nextRequest(request))!=null) {
      // update or receive a new flowgraph
      source    = request.getSource(source);
      flowgraph = request.getFlowgraph(flowgraph);
      scheduler = request.getScheduler(scheduler);
      frameID   = request.getResponseId();

      ResponseFrame responseframe = new ResponseFrame();
      responseframe.init(flowgraph, scheduler, 0);
      responseframe.setSessionID(this.fSessionID);
      responseframe.setDataID(frameID);
```

```
24:      vtkDataSet data = source.getDataFrame();
25:      responseframe.setData(data);
26:      server.send(responseFrame);
       }
27:    state.deleteObject(fSessionID);
28:    return null;
     }
   }
```

The handler's `execute` method initializes the request queue, adding the first request to the queue. The handler is registered in the server's soft storage (line 8), allowing later request frames to get a reference to the handler. If an initial request is included in the request handler, this request is processed right after the initialization, otherwise the request handler waits for requests from the client. The handler obtains the next request from the queue (line 5) and creates a new response frame with the appropriate flowgraph and scheduler (lines 16-23). The data for the response frame is read from the source (line 24). The response frame is sent to the next host using the `send` method defined in the server's interface. The handler processes requests until it receives a end-session request or timeouts waiting for a request (`nextRequest` method, line 15). When there are no more requests to process the handler is unregistered from the server's state (line 27) and a null host address is returned from the execute method (line 28) causing the frame to be discarded after its execution.

6.4.2 Request frames

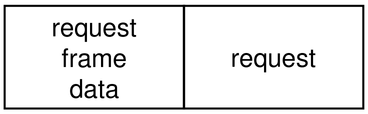

Figure 15: Request handler frame

Figure 15 shows the structure of a `RequestFrame` object. `RequestFrame` is an active frame that carries a request from the client to the server.

```
public class RequestFrame implements ActiveFrame {
  private Request request;

  public HostAddress execute(ServerState state)
    throws FrameException
   {
    RequestHandler rserver
          = (RequestHandler)state.getObject(this.sessionID);

    if (rserver != null) {
      rserver.addRequest(this.request);
    }
    return null;
   }
  }
```

The `execute` method finds the the registered request handler for the session (lines 4-5). The request carried by this frame is added to the request queue (lines 6-7). The `addRequest` method handles the queue including whether the passes parameter is an overriding request or not. This method always returns a null host address (line 8) and the frame is discarded after execution.

6.4.3 Response frames

response frame data	visualization flowgraph	scheduler	visualization dataset

Figure 16: Response handler frame

Figure 16 shows the structure of a `ResponseFrame` object. The request handler responds to requests by sending response frames back to the client. Response frames are specialized active frames that contain a visualization flowgraph, the data to visualize and a scheduler. The response frame may travel through many frame servers on its way back to the client. At each server along this path, the response frame executes parts of the visualization flowgraph. Here is a simplified version of the response frame:

```
 1:public class ResponseFrame implements ActiveFrame {
 2:   Flowgraph flowgraph;
 3:   Scheduler scheduler;
      ...
      // create and execute a part of the flowgraph in this host
 4:   public HostAddress execute(ServerState state)
 5:     throws FrameException
      {
        // get the data receiver
 6:     DataFrameReceiver dataReceiver =
 7:       (DataFrameReceiver)state.getObject("data-receiver");
 8:     vtkDataSet dataset    = dataReceiver.getDataSet(this.sessionId,
 9:                                                   this.dataId);

10:     DvHostAddress nextHost;
11:     HostAddress localHost = state.getLocalAddress();
12:     int nofnodes          = this.flowgraph.getNumberOfNodes();

13:     do {
14:       dataset=this.flowgraph.buildNode(this.currentNode,currData);

15:       if (dataset != null) {
16:         dataset.Update(); // execute the node
          }
17:       this.currentNode++;
18:       nextHost = this.scheduler.getHost(this.currentNode);
19:     } while (this.currentNode < nofnodes && nextHost != null
20:              && nextHost.equals(localHost));
```

```
21:      if (nextHost != null) {
22:        DvFrameServer server = (DvFrameServer)state.getServer();

           // send the data to next host
23:        server.SendData(dataset,nextHost,this.sessionId,this.dataId);
         }
         // return the next hop address
24:      return nextHost;
       }
    }
```

The `ResponseFrame` implements the `ActiveFrame` interface (line 1). The response frame carries with it the visualization flowgraph to execute (line 2) and a scheduler (line 3). The scheduler determines where the different nodes of the flowgraph are executed. The response frame implements the `execute` method declared in the active frame interface (line 4). In lines 6-9 a handle to the dataset associated with this response frame (session/data id tuple) is retrieved from the data receiver. In lines 10-20 nodes of the flowgraph are built and executed as determined by the scheduler (line 18). In line 14 the current node of the flowgraph is built and then in line 16 the node is executed. In line 18 the scheduler is asked for the address of the host to execute the next node in the flowgraph. If the scheduler determines that the next node should be executed in the same node then the process is repeated (back to line 13). Otherwise the dataset is sent to the next host using the `SendData` method described in 6.1 (lines 21-23). Finally the address of the next host is returned from the method (line 24) and the server will send this response frame to that address if not null.

7 Scheduling of active frames

The scheduling decision for a frame is delegated to the particular implementation of the scheduler interface described in section 6.3.4. The scheduler is transmitted along with the frame program. The `getHost` method is called at every server the frame visits. The scheduler can make decisions at different points of the frame's lifetime. This mechanism permits the implementation of various scheduling policies, allowing the particular implementation to adopt the strategy most appropriate for the application.

The scheduler can use application-specific information to make scheduling decisions [5]. In the presence of resource management systems like Darwin [30] or Globus [13] the scheduler can use its "knowledge" of the application to reserve the required resources. In environments where no resource reservation is supported, like a best-effort internet, schedulers can combine applications-specific information with resource availability data to make scheduling decisions. Resource availability data and predictions can be obtained from systems such as Remos [24] or NWS [35]. Here are some examples of scheduling scenarios ranging from a completely static one to the most dynamic one.

Scheduling at service deployment time: In this scenario the hosts where the application executes are fixed except for the client host. The main use of this approach is to satisfy well-known application constraints. This approach can also be used in environments where a single instance of the application has exclusive access to the resources. A system administrator can use his knowledge of the application requirements and execution environment to determine the set of hosts the application will use. Also an off-line scheduling algorithm can be used to find the optimal mapping. The scheduling process does not take into account the client host, limiting the adaptation to client heterogeneity. Obviously, since this approach does not take into account resource availability information it cannot adapt dynamically to resource changes. Even with this rigid approach, the application obtains the benefits of resource aggregation. It is also worth mentioning that this particular approach is useful for application and scheduling debugging.

Scheduling at session startup: In this scenario the application uses the same host assignment for all frames of the same type. Each frame visits the same set of servers and executes the same set of calculations. The scheduler can exploit information about the application resource demands and the client capabilities to create the schedule at the beginning of a session. This increases the level of adaptation to client heterogeneity. However it cannot adapt to dynamic resource availability. The scheduler can choose from a pool of compute resources the hosts needed to execute the application. If the duration of a typical session is relatively long, then it makes sense to use long term resource information to make a scheduling decision. A particular implementation could interact with a resource management system, if available, to make resource reservations. The following example shows the implementation of this type of scheduler:

```
 1:class SessionScheduler implements Scheduler {
 2:  StaticScheduler(HostAddress[] list) {
 3:    this.hosts = list;
     }
 4:  HostAddress getHost(int hopCount) {
 5:    if(hopCount<numberOfHosts) {
 6:      return this.hosts[hopCount];
       }
 7:    return null;
     }
   }
   ...
 8:HostAddress[] list = getHostList();
 9:Scheduler sched    = new SessionScheduler(list);
10:ActiveFrame frame  = new SampleFrame(sched);
11:server.send(frame);
```

The application computes the list of hosts to execute once (line 8) and then uses that set of hosts for all frames. The scheduler carries the list of host addresses and returns the appropriate address from that list.

Scheduling at request creation: In this scenario, a new schedule is computed for each request. However all response frames follow the same path and execute the same set of calculations. The scheduler can combine application-specific information with resource availability data when creating the schedule, adding some degree of adaptation.

Scheduling at frame creation time: In this scenario, the frames are scheduled at the source server. The scheduler can use resource availability data when scheduling a new frame. This scheme provides a higher degree of adaptivity to the application, because different frames may follow different paths and use different compute resources. However, since the schedule for any individual frame is static, this adaptation only occurs at the source. In the following example the scheduler generates the path of compute hosts at the source when the `hopCount` is 0 (line 3) and uses that path for the frame transmission.

```
1:class AtCreationTimeScheduler implements Scheduler {
   ...
2:  HostAddress getHost(int hopCount) {
3:    if(hopCount==0) {
4:      this.hosts = computePath();
      }
```

```
    if(hopCount<numberOfHosts) {
      return this.hosts[hopCount];
    }
    return null;
  }
}
```

Scheduling at frame delivery time: In this scenario, the application can make scheduling decisions just before the active frame is delivered to the next server. The application can react to changing resource conditions even if the frame is in flight. This offers the highest degree of adaptivity, but might introduce significant overhead depending on the complexity of the decision making procedure.

```
class AtDeliveryTimeScheduler
    implements Scheduler {
 HostAddress getHost(int hopCount) {
    if(hopCount<numberOfHosts) {
      return getHostNow();
    }
    return null;
  }
}
```

The previous scheduling scenarios schedule the frames at a single point, making it easier to achieve global coordination. With scheduling at frame delivery time it is more difficult to coordinate the execution of multiple frames that are traveling through the system.

8 Putting it all together

In the previous sections we have described the different components needed to build the remote visualization services. In this section we summarize the process of using active frames to provide higher level services like the remote visualization service. We also summarize the process of building a specific visualization service using the Dv library.

8.1 Building a service with Active Frames

In order to build a heavyweight service like Dv it is necessary to identify the application model that better suits the service we want to provide and how the computation and other resources can be partitioned. For example, in the case of Dv, many visualization applications are structured as a flowgraph of independent filters that transform the data into images (See Figure 10). For this application model, it is very easy to partition the application flowgraph into independent filters. Dv divides the application into data source objects, visualization flowgraphs and requests. The process to build a visualization service using the Dv framework is described in the section 8.2.

The next step is to create a set of active frames to support the most common tasks for the service. For example, Dv defines three types of active frames: `RequestHandlerFrame`, `RequestFrame` and `ResponseFrame`. The `RequestHandlerFrame` keeps state on behalf of the client at the server where the dataset is located. This frame also handles subsequent requests from the client. The `RequestFrame` carries the requests from the client to the remote server. `ResponseFrames` are generated in response to client's requests.

The next step is to extend the frame servers with the service-specific libraries that contain the routines more commonly used by the frames defined for the service. In the Dv framework, the servers are extended with visualization libraries and helper functions to transfer datasets between hosts. The servers are extended using the mechanisms explained explained in section 3.3.2.

Dv provides an interface that enables the use application-level information in resource selection decisions. The application can provide its own implementation of the scheduler interface or it can use one of the default schedulers provided by Dv.

8.2 Building a remote visualization application with Dv

The Dv framework divides the application into three components: Data source, visualization flowgraph and request objects. A particular visualization application provides the specific implementation of these objects.

The data source object is used to access the dataset. This object understands the format and mechanism used to store the dataset. The data source produces a dataset object that can be used with the routines included in the visualization libraries. The data source object can be reused in different visualization applications that access the same dataset.

The visualization flowgraph specifies the transformations to perform on the dataset. This object is the core of the visualization process. This object also specifies how the visualization process can be divided into individual tasks, for instance each visualization filter can be executed individually. The flowgraph object is specific to a particular visualization for a dataset. However, the parameters used to configure and execute the flowgraph are specified by the user at runtime.

The request objects carry the parameters for the flowgraph. These parameters modify the way the visualization process behaves. For instance the request contains parameters like the region of interest, mesh resolution and isovalues. The request objects are strongly tied to the flowgraph, since the requests carry flowgraph-specific parameters.

9 Evaluation

The active frame mechanism is clearly general and flexible. However, a potential disadvantage is that this flexibility introduces performance overhead. Additional space, and thus bandwidth, is required by the frame program, and extra processing time is necessary to interpret and execute the frame. The applications in which we are interested require significant amount of resources to process and transfer the large amount of data they operate on. For these resource-intensive applications, we expect the overhead associated with processing the frame program to be relatively small compared to the resources required to process the application data. In this section we begin to evaluate the costs and benefits of this flexibility.

The evaluation was done using a relatively small dataset as input, thus making it more difficult to amortize the cost of processing Active Frames. The input dataset for this evaluation was produced by the `183.equake` benchmark program in the SPEC CPU2000 benchmark suite [27, 19]. The input dataset contains ground motion data with 30K unstructured mesh node and 151K tetrahedral elements. The size of each frame data is 120 KB. The dataset contains 165 frames, each containing a scalar value for the horizontal displacement of each node in the mesh. The region of interest specified for the animation includes the complete volume of the basin and all the frames in the dataset.

9.1 Single-host setup

We want to obtain a baseline of the time elapsed to satisfy the user's request and the elapsed time for each stage of the visualization flowgraph. Additionally, we want to know whether it is possible to obtain accept-

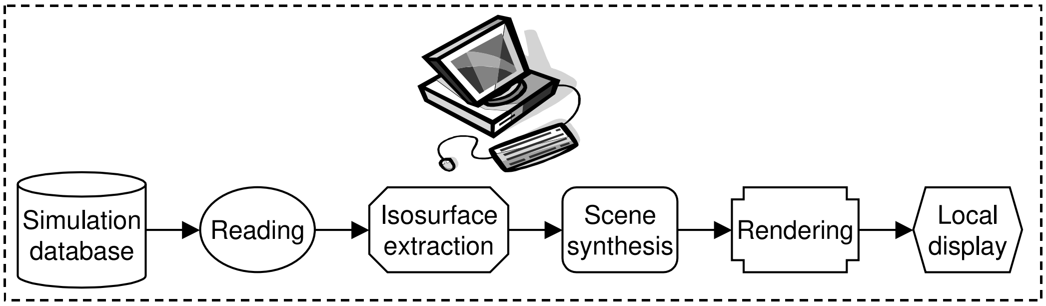

Figure 17: Single host setup.

able response times when visualizing the dataset using only the resources available to the client machine.

This section characterizes the execution time on a single host of the visualization described in Section 5. The measurements shown below were taken on a Pentium-III/450MHz host with 512 MB of memory running NT 4.0 with a Real3D Starfighter AGP/8MB video accelerator.

Isosurfaces Operation	10 time (ms)	20 time (ms)	50 time (ms)
Read frame data	584.35	582.65	586.13
Isosurface extraction	1044.37	1669.73	3562.36
Rendering	55.72	63.73	125.77
Total	1684.44	2316.11	4274.26
Frames per second	0.59	0.43	0.23

Figure 18: Mean execution time (local)

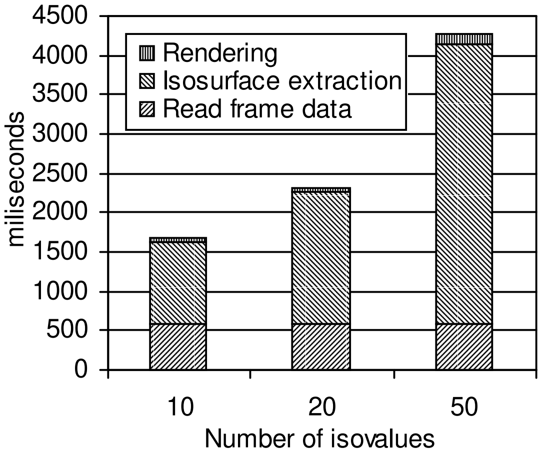

Figure 19: Execution time breakdown (local).

Figure 18 summarizes the cost of running a purely sequential C++ version of the visualization on a single host. The application pays an average one-time cost of 5.389 seconds to load the mesh topology (not shown in table). The application caches the mesh information since it is the same for all frames. To process a frame, the application spends an average of 583.78 ms loading the data for each frame.

Figures 20 and 21 show the execution time of isosurface extraction and rendering. Execution time for

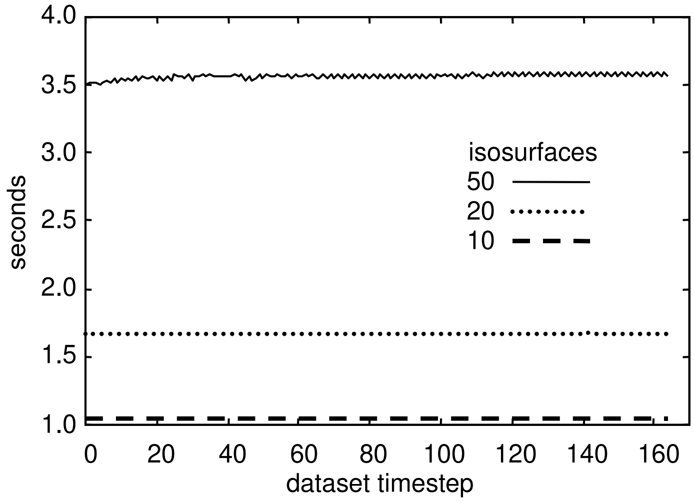

Figure 20: Isosurface extraction (local).

each tasks depends on the *number-of-isosurfaces* parameter (See Figure 18). The isosurface extraction task produces a more complex output dataset with finer granularity and a greater number of polygons as the number-of-isosurfaces parameter increases, which in turn slows down the rendering task. Figure 21 shows that the execution time of the rendering task is affected by the content of the dataset. During the initial time steps only the ground close to the epicenter is moving. As time passes the wave propagates to the surface, which affects a larger region of the dataset. Toward the end, the effect of the wave is dampened and only the ground near the surface is still shaking.

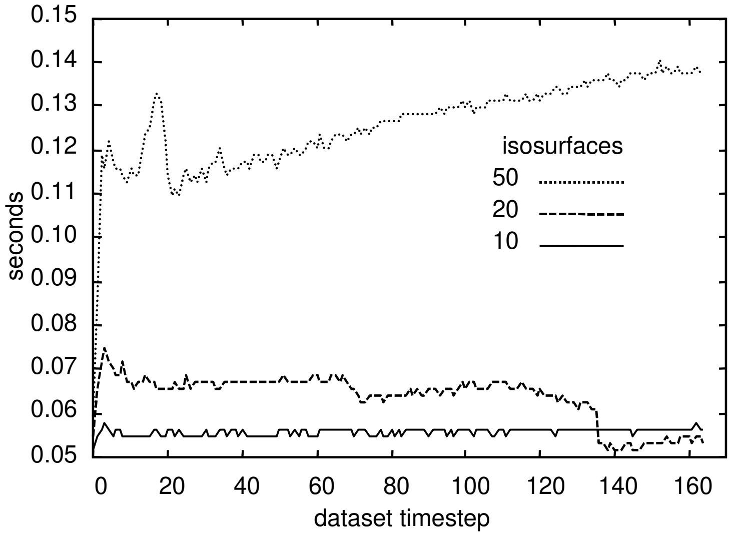

Figure 21: Rendering (local).

These results suggest that (1) additional resources are needed; and (2) expected execution time of a task can be controlled by varying application parameters. In order to achieve interactivity, the application must be responsive to the user's input. The total time required to process a single frame (See Figure 18) is too large for interaction. By varying application parameters such as number-of-isosurfaces, the application can either produce a higher quality version of the visualization when enough resources are available, or it can reduce the frame's processing time to meet the frame's deadline.

9.2 Pipeline setup

We wanted to find out (1) the cost of processing Active Frames in the remote visualization service and (2) whether adding more compute resources to satisfy user requests would actually decrease response time and increase the number of frames that can be processed per second.

To try to make the animation more responsive, we can switch (*without changing the application source code*) from the single-host setup to a three-host setup. Two of these hosts are used as compute servers and the other as a display client. The client host is the same as in Section 9.1. The two additional compute servers are 550 MHz Pentium III processor machines with Ultra SCSI hard disks running version 2.0.36 of the Linux kernel. The server hosts are connected using a 100 Mbps switched Ethernet. The client communicates with the servers through a 10Mbps Ethernet.

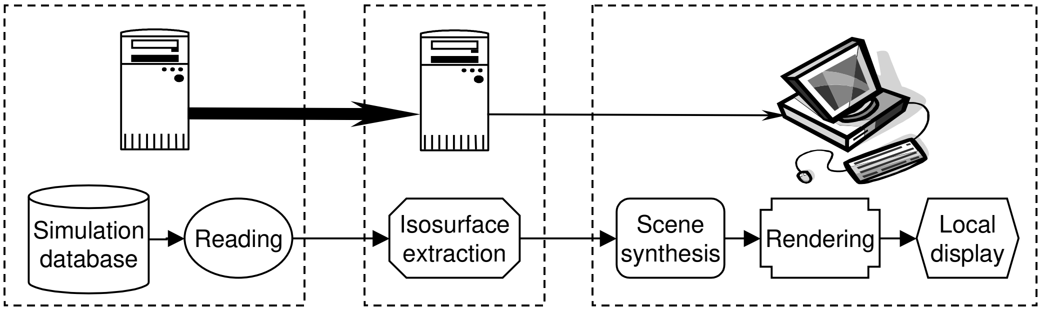

Figure 22: Pipeline setup

The scheduler used in this configuration assigns the tasks to the three hosts as follows (See Figure 22): One of the hosts contains the dataset to visualize. The other compute host performs the isosurface extraction. The client host performs the remaining tasks, including scene generation and rendering to the screen.

# Isosurfaces Operation	10 time (ms)	20 time (ms)	50 time (ms)
Read frame data	286.21	285.98	287.11
program transfer(s-s)	110.14	94.47	86.51
Data transfer(s-s)	118.88	116.92	119.52
Isosurface extraction	696.43	1152.03	2561.41
Program transfer(s-c)	39.54	44.13	46.40
Polys transfer	175.10	435.66	1700.59
Render	64.99	162.32	340.40
Total	1491.29	2291.51	5141.95

Figure 23: Mean elapsed time (pipeline)

Figure 23 shows the time required to complete each of the operations, including the additional transfer of the data and program from the source to the compute server (marked as s-s) and from the server to the client (s-c). Note that the transfer of the program is significantly slower from the source to the compute server than from the compute server to the client. The compute server becomes the bottleneck of the execution (See Figure 25) and the task receiving the frame must compete for compute resources to demarshall the frame program and data. The time to transfer the data from the compute server to the client is significantly higher

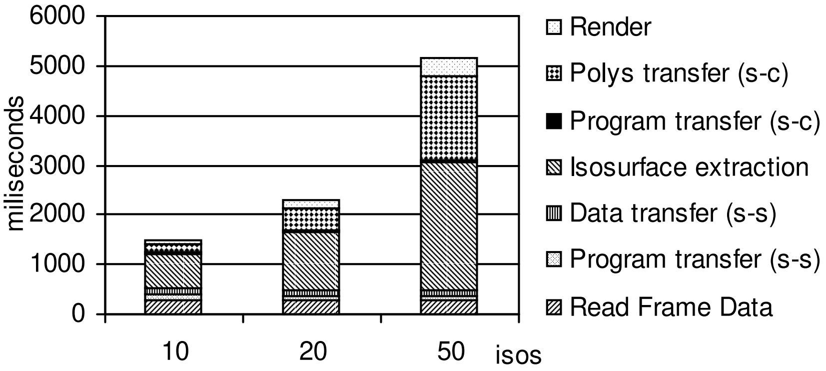

Figure 24: Execution time breakdown (pipe).

for the extraction of 50 isosurfaces because the polygonal structure is much more detailed and complex in this case, requiring more bandwidth.

	Processing time (ms)			Frames per second		
# isos	10	20	50	10	20	50
Source	515.23	497.37	493.15	1.94	2.01	2.03
Server	1140.09	1843.21	4514.44	0.88	0.54	0.22
Client	279.63	642.11	2087.40	3.58	1.56	0.48
Max frame rate				0.88	0.54	0.22

Figure 25: Mean frame rate (pipeline)

Despite the additional cost of transferring the data and the program, it is possible to obtain the benefits of resource aggregation. Figure 25 shows the time each host spends processing a single frame and the respective frame rate each host can produce. For all cases, the compute server has the slowest frame rate, which determines the overall frame rate of the pipeline. Another significant advantage over the local setup is that the client now has plenty of spare cycles for interaction with the user (i.e., Rotate, zoom-in).

Figure 26 shows time to the render a frame at the client throughout the animation. The rendering of a frame is slower and more variable in this setup than in the single-host setup because the rendering task shares compute resources with the frame transfer task.

9.3 Fan setup

In the previous setup (Section 9.2), the isosurface extraction stage of the flowgraph becomes a bottleneck. We wanted to know (1) whether adding resources would improve the performance of the visualization and (2) how this new resource configuration would affect time to execute each flowgraph stage and the cost of processing Active Frames.

In this setup, two additional compute servers, with the same specifications as described in the previous section, are used to execute the application. For each frame, the source server reads the dataset and the client generates and renders the scene. The scheduler uses the remaining servers to execute the isosurface extraction by sending each server a different frame in a round-robin fashion (See Figure 27). The frames are collected, reordered if necessary, and rendered by the client.

Figure 28 shows that the client host is saturated, which causes the whole system to slow down. In this case both the program and data transfer operations are slowed down at the client (See Figure 29). However,

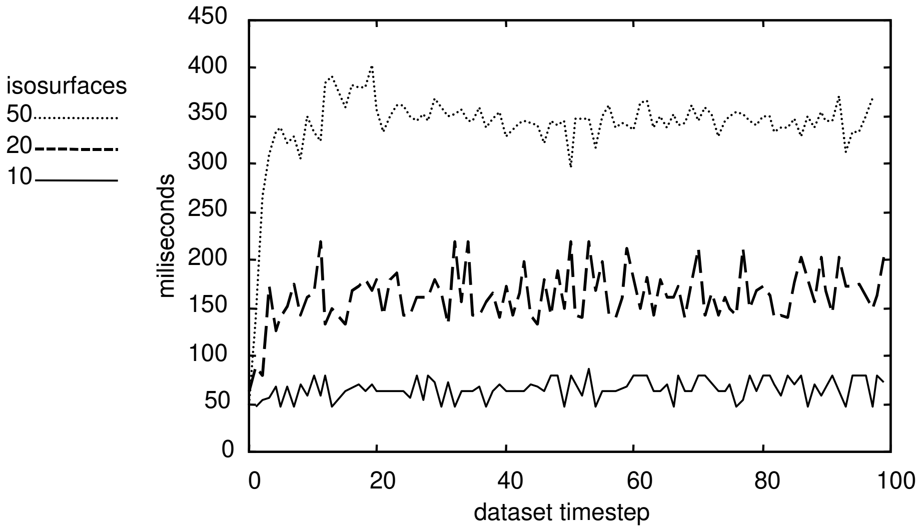

Figure 26: Render (pipeline).

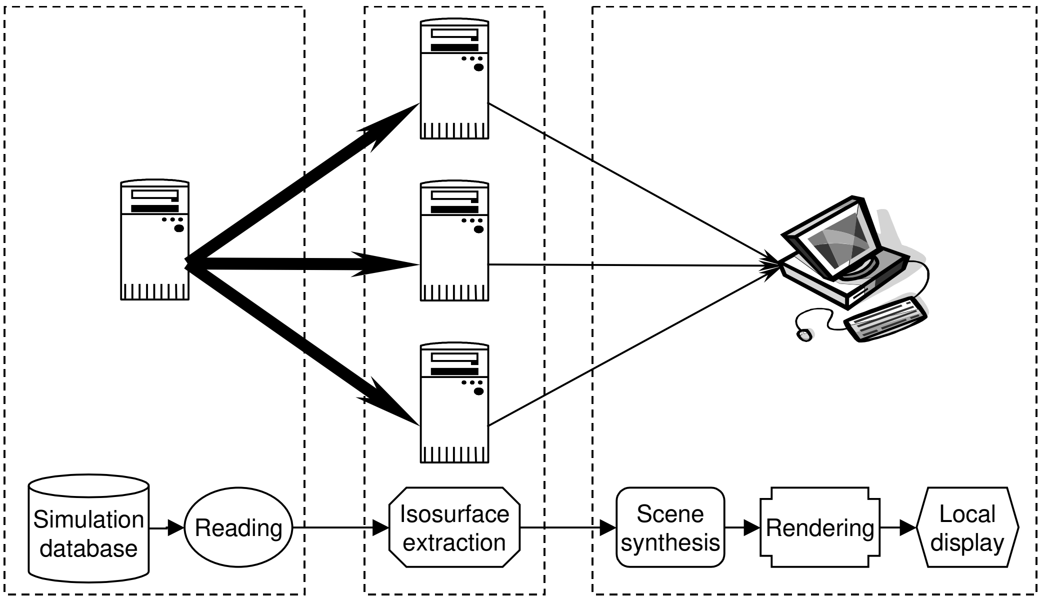

Figure 27: Fan setup

# isos	Processing time (ms)			Frames per second		
	10	20	50	10	20	50
Source	413.43	414.85	418.24	2.42	2.41	2.39
Server x 3	1209.77	1746.87	4590.96	2.48	1.72	0.65
Client	525.96	701.03	2345.04	1.90	1.43	0.43
			Max frame rate	1.90	1.43	0.43

Figure 28: Mean frame rate (fan)

by using application-specific information in the scheduler, the total frame rate is increased by a factor of 3 in two out of the three cases and double in the other case, compared to the single-host setup.

Isosurfaces Operation	10 time (ms)	20 time (ms)	50 time (ms)
Read data	283.46	283.58	284.52
Program transfer(s-s)	13.61	15.09	17.73
point data transfer	116.36	116.18	115.99
Isosurface	677.09	1098.01	2384.19
program transfer(s-c)	65.26	83.47	173.76
polys transfer	337.45	434.13	1899.29
Render	123.24	183.44	271.99
Total	1616.47	2213.88	5147.48

Figure 29: Mean execution time (fan)

The rendering time is increased even further (See Figure 29) and exhibits higher variability (See Figure 30), because the client host is under high load, which causes the rendering task to be preempted more often.

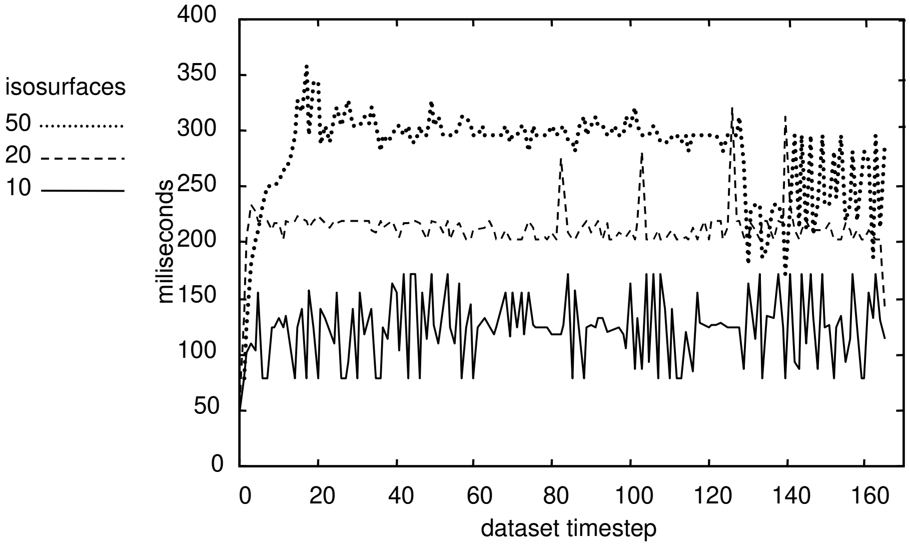

Figure 30: Render (fan).

10 Barriers to acceptance

This section describes currently what the main limiting issues are in using active frames to deploy remote visualization and other heavyweight services. The discussion is divided in three topics. The first topic discussed in section 10.1 is about the limitations of the current implementation. Section 10.2 discusses the security issues of the system. The last section 10.3 points the direction for future research.

10.1 Current limitations

The current implementations of the active frames mechanism and Dv framework are not optimized. Two common optimizations techniques used to improve the performance of Internet services are connection pooling and thread pooling.

Connection pooling: After a frame is transfered between servers, the servers could keep the connection open for a given period of time instead of shutting it down. The next time a frame is forwarded between these two servers, the sending server would check its connection pool. If a connection between the two servers exists and it is in good state (i.e., it has not timed out) then that connection could be reused. This approach would reduce the total time to transfer a frame when the latency between the involved pair of servers is relatively high. However, if the latency between the servers is low this approach would not provide any benefit and it might actually hurt performance. Another caveat of this approach is that it is not completely clear what is a good policy to manage the connection pool. If too many idle connections remain open, they consume valuable system resources like connection descriptors and buffer space.

Thread pooling: The frame servers use the standard approach of creating a new thread to process a new frame that arrives. A server could create a pool of threads ahead of time to process the incoming frames. This approach eliminates the thread creation time. Although at first it seems like this mechanism would reduce the frame processing time, we chose not to implement it since the thread creation does not contribute significantly to the total processing time of a frame.

10.2 Security issues

Active frames provide a flexible mechanism to deploy new services or customize existing ones. However, this flexibility introduces various security threats to the grid of host executing the active frames. Even though we do not address the security issue completely, we impose various restrictions to the dynamic code loading mechanism and the execution of the active frames. These restrictions are implemented using Java's security facilities. For example, the dynamic code can be loaded only from a set of trusted code servers. The application libraries have to be located in the same host where the frame server executes and they are pre-loaded when the frame server starts execution. Active frames call the methods declared in the frame server interface. Additionally, the active frames can call the methods defined in the pre-loaded application libraries as described in the previous section. However, this does not prevent a faulty library from crashing the frame server. More elaborated techniques, like code signing [16], software fault isolation [34] or proof carrying code [25] could be used in this case.

Resource management is probably the most difficult task in building a safe execution environment. The frame servers do not restrict on the amount of resources consumed by a frame. For instance a misbehaved frame could consume an unbounded number of compute cycles within one server or across multiple server. In this case other frames are able to make progress much more slowly than they normally would. In general, we expect the underlying resource managers to take care of the access to the resources. Reservation systems at the network level like Darwin [30] will help to limit the amount of bandwidth a misbehaved frame can consume.

10.3 Future directions

Active frames provide a mechanism for heavyweight services to send computation to other compute hosts. The scheduler interface allows for the use of application-level information to select the resources used by the application. In order to support resource-aware services completely the framework must provide: (1) information about the resource availability, (2) information about the resource requirements of the application and (3) a mechanism to automate the adaptation process.

Resource availability: The application needs resource information ahead of time to be able to adapt. Then the application could choose between two or more resources based on availability and overall cost. Moreover, the service not only needs information about the current resource availability, but also about the availability of those resources while the service will be using them in the near future.

If the underlying manager for a particular supports reservations then it is assumed that the manager will perform the appropriate admission control when the resource is requested. It is also assumed that the resource manager will guarantee the availability of the resource for the requested period.

Unfortunately, many resources work under a best-effort model and seldomly provide information about the current resource availability. In this case it is necessary to measure the current load of a resource and also predict its future availability. This information can be obtained from systems like CMU REMOS (REsource MOnitoring System) [24] or NWS (Network Weather Service) [35]. We plan to leverage the work done in these projects. Initially the REMOS implementation will be integrated in the framework. The current release of REMOS provides information and prediction about network bandwidth and delay between a given set of nodes. REMOS also provides information about the future load of a compute host based on prediction.

Application resource requirement: In order to select the appropriate resources to provide a service, the application first needs to know the resource requirements to complete a task. In general, the resource requirements of the applications are not well understood. Moreover, the resource requirements vary during the application execution as the application goes from one stage or mode to another or as a result of a parameter or input change.

We think that an approach that involves both system support and collaboration from the application could prove useful to determine the resource required to complete a task. The system could provide mechanisms to measure and keep a history of the resource consumption for a particular task. The application has to provide the system with information about the tasks the application executes like the starting and ending points and the parameters used to execute the task. The system could then use the collected data to characterize the resource usage of the individual tasks in the application.

Automatic adaptation: Given the information about the available resources and tasks' resource consumption it is desirable to automate the adaptation process for the application. The goal of the adaptation process is to improve the user's perceived experience of the application. The motivation to automate this process is that in general computers are very efficient and fast at processing large amounts of data and detecting patterns in noisy data. There are two different types of adaptation approaches we think would be useful for this kind of applications: (1) resource selection and (2) quality level selection.

Resource selection: The scheduling interface defined in the Dv framework is a powerful mechanism that enables the the use of application-level information for resource selection. So far, we have used the interface to select resources statically before the application starts executing. The user specifies the mapping from tasks to compute servers. This provides a very basic form of adaptation where the knowledge of the programmer about the application and the user's intuition are used to select resources at "load time". However this approach is error prone and does not take into account dynamic resource information.

The next logical step is to create "smarter" implementations of the application-level schedulers that use resource information to automate the selection of resources. These schedulers would take as input the resource requirements for the application tasks and a set of available resources and would select the resources most appropriate to execute the tasks. An important point to keep in mind is that these schedulers have to make a decision very quickly to satisfy the time constrains. In this case a "good" but suboptimal answer on time is much more important than an optimal resource selection that is late. Remember also, that the application-level scheduler is not doing resource management or scheduling per se, it is just selecting a good combination of resources to execute the desired tasks. It is up to the underlying manager to arbitrate the access to the resources according whether it is best-effort or guaranteed service.

Quality level selection: In this approach, the objective is to find a set of application parameters that would allow the tasks to execute with the available resources. The idea behind this approach is that there is a reduced set of parameters that influence significantly the resource consumption of a particular task. For example, it takes longer to compute 20 isosurfaces from a particular dataset than to compute only 5 isosurfaces on the same dataset. The application-level schedulers would take as input information about the available resources and a description of resource consumption according to task parameters. The schedulers would produce a set of task parameters to execute the application.

These two approaches are not mutually exclusive. It would be possible to try to select the most appropriate resources to execute the tasks and at the same time vary the application parameters. However, this might become intractable very fast. A simpler approach would be to select resources with a fix set of parameters, and only try to change application parameters when the available resources are not sufficient to complete the desired set of tasks. At this point, a candidate set of resources is fixed and the scheduler simply searches for the parameters that would allow the tasks to execute with that set of resources. In general, we expect the user to choose the application parameters. These parameters would only be modified if there are not enough resources to execute the tasks.

11 Related work

The idea of bundling programs with network data is certainly not new to active frames. This idea was exploited effectively by active messages in the context of parallel processing [33], and by active networks in the context of adding additional functionality to network routers [31]. Active services and proxies [2, 15] advocates for the placement of user defined computation within the network without perturbing the network layer implementing value-added services at the application level.

Similarly, process level migration is supported in systems like Condor [23] and Sprite [11] which intend to migrate processes to load balance a cluster of workstations. Fine-grained application controlled mobility is supported in Emerald [20].

The idea of extending the runtime environment dynamically is central to Java [18, 22]. Dv's implementation relies heavily on Java's dynamic class loading mechanisms. Systems like HARNESS [10] provide plug-in interfaces that allows applications to extend the runtime's features.

ABACUS [3] supports automatic function placement in the context of data-intensive applications in system and local area networks. Odyssey [26, 28] supports adaptation for applications in the context of mobile computing. Odyssey provides system-level support on the mobile host to achieve the best overall quality level for the applications running at the mobile host. The system monitors and manages the available resources at the mobile host. The system also gives the application feedback about the available resources and the quality level at which the application should run.

The use of application-level information to make scheduling decisions in distributed systems is successfully implemented in AppLeS based systems [5]. The scheduling interface defined by the Dv framework is a type of application-level scheduler.

Customizable resource management mechanisms like Darwin [30] allow applications to combine bandwidth and other network resources at the end points with resources inside the network to deliver enhanced quality services to the user. Active frames complements this idea by allowing the use of applications-specific information to combine bandwidth and compute resources.

12 Conclusions

Heavyweight services, such as remote visualization, often require more resources than the ones available to the client requesting the service. We proposed a model where resources of multiple compute hosts are aggregated to satisfy the user's request. We have shown it is possible to provide services with this model. We built a remote visualization service that takes advantage of the proposed resource model. Active Frames is the key mechanism to exploit this resource model, allowing applications to move work and data to the hosts in the remote and local sites. Our evaluation shows that the cost of processing active frames is relatively small for heavyweight services.

Information about the application's resource consumption is important to to successfully exploit available resources. The scheduling interface permits the use of application-level information to select resources. So far, we have used the scheduling interface to statically select the resources to execute our visualization service. Using this mechanism we are able to run our visualization service under different resource configurations. However, we did not use dynamic information about application resource consumption and resource availability.

The result of our experiments show that the resource consumption of our visualization service varies considerably according to runtime parameters like resolution, region of interest and number of isosurfaces to visualize. This suggests that the runtime parameters can be used as information to determine application resource consumption thus improving the resource selection process. However, further research is needed to find how to use the runtime parameters to characterize the resource consumption of the application.

In our evaluation, the remote visualization service executed on isolated resources, thus no dynamic information about resource availability was used. However, in order to execute the service with shared resources the resource selection process should take into account dynamic information about resource availability.

We created the Dv library, to build our remote visualization service by grid-enabling existing visualization packages. Dv provides the interface to access the existing visualization library and the routines to transfer visualization datasets between hosts.

References

[1] Martin Aeschlimann, Peter Dinda, Loukas Kallivokas, Julio Lopez, Bruce Lowekamp, and David O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.

[2] Elan Amir and Steven McCanne adn Randy Katz. An active service framework and its application in real-time multimedia transcoding. In *SIGCOMM'98*, pages 178–189. ACM, Sep 1998.

[3] Khalil Amiri, David Petrou, Greg Ganger, and Garth Gibson. Dynamic function placement in active storage clusters. Technical Report CMU-CS-99-140, Carnegie Mellon University, 1999.

[4] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O'Hallaron, J. Shewchuk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 152:85–102, January 1998.

[5] Francine Berman and Richard Wolski. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96*, pages 100–111, August 1996.

[6] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical Report CS-95-313, University of Tennessee, Nov 1995.

[7] Gray C.G. and D.R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Ariz., 1989. ACM.

[8] Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java[tm] Class Libraries. Supplement for the Java[tm] 2 Platform, Standard Edition, v1.2*, volume 1 of *Java series*. Addison-Wesley, 2 edition, 1999.

[9] P. Dinda, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. The case for prediction-based best-effort real-time systems. In *Proc. of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS*

1999), volume 1586 of *Lecture Notes in Computer Science*, pages 309–318. Springer-Verlag, San Juan, PR, 1999.

[10] Jack Dongarra, Al Geist, James Kohl, Philip Papadopoulus, and Vaidy Sunderam. HARNESS: Heterogeneous adaptable reconfigurable networked system. In *7th Symposium on High Performance Distributed Computing HPDC'98*, Chicago, Illinois, July 1998. IEEE Computer Society.

[11] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software – Practice & Experience*, 21(8):757–785, Aug 1991.

[12] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. Number 0764580434. IDG Books, 1998.

[13] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[14] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computating Infrastructure*. Morgan Kaufman, 340 Pine Street, Sixth floor, San Francisco, CA 94104-3205, 1999.

[15] A. Fox, S. Gribble, E. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, Cambridge, MA, October 1996. ACM.

[16] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, chapter 4. Addison-Wesley, 1999.

[17] Rob Gordon. *Essential JNI: Java Native Interface*, volume JNI of *Essential*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1998.

[18] J. Gosling, Joy W., and Steele G. *The Java Language Specification*. Number 0201634511. Addison-Wesley, Menlo Park, California, 1996.

[19] J. Henning. Spec2000: Measuring cpu performance in the new millenium. *IEEE Computer*, pages 28–35, July 2000.

[20] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Jan 1988.

[21] Sheng Liang. *The Java[tm] Native Interface: Programmer's Guide and Specification*. Java series. Addison-Wesley, 1999.

[22] T. Lidholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 2 edition, 1999.

[23] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[24] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symp. High-Performance Distr. Comp.*, jul 1998.

[25] G.C. Necula and P Lee. The design and implementation of a certifying compiler. In *PLDI'98 Conf. on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[26] B. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[27] David R. O'Hallaron and Loukas F. Kallivokas. The SPEC CPU2000 183.equake benchmark. www.spec.org/osg/cpu2000/CFP2000/, 2000.

[28] M. Satyanarayanan and D. Narayanan. Multi-fidelity algorithms for interactive mobile applications. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Seattle, WA, August 1999.

[29] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1998.

[30] Peter Steenkiste, Allan Fisher, and Hui Zhang. Darwin: Resource management in application-aware networks. Technical Report CMU-CS-97-195, Carnegie Mellon School of Computer Science, December 1997.

[31] David Tennenhouse and David Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, August 1995.

[32] C. Upson, T. Faulhaber, D. Kamins, et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[33] T. Von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th Int. Conf. on Computer Architecture*, pages 256–266, May 1992.

[34] R. Wahbe, S. Lucco, and T. Anderson. Efficient software-based fault isolation. In *14th Symposium on Operating Systems Principles, SOSP'93*, pages 203–216. ACM, Dec 1993.

[35] Rich Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC97)*, pages 316–325, August 1997. extended version available as UCSD Technical Report TR-CS96-494.