Run-time Support for Adaptive Heavyweight Services

Julio C. Lopez and David R. O'Hallaron
{jclopez,droh}@cs.cmu.edu

Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. By definition, a heavyweight network service requires a significant amount of computation to complete its task. Providing a heavyweight service is challenging for a number of reasons. First, since the service cannot typically be provided in a timely fashion using a single server at the remote site, multiple hosts at both the remote and local sites must be employed. Second, the available compute and network resources change with respect to time. Thus, an effective service must be adaptive in the sense that is able to transparently aggregate the available resources and react to the changing availability of these resources. In this paper we present a framework that allows us to build these kinds of adaptive heavyweight services. Experimental results with a distrubuted visualization service suggest that the cost imposed by the new capability is reasonable.

1 Introduction

Existing Web-based network services are typically *lightweight* in the sense that servers do a relatively small amount of work in order to satisfy each request. This is primarily due to concerns about minimizing response latency for clients and not overloading the compute resources of high-volume servers. If these concerns were addressed, service providers could also offer *heavyweight* services that require significant amounts of computation per request. Examples include search engines with advanced IR algorithms, data mining, and remote visualization of large datasets.

Providing heavyweight services is challenging for a number of reasons: (1) Service providers can only contribute so many compute resources, and often these resources will be insufficient; (2) For a given service, the available compute and network resources are often quite different, and can also change over time. For example, different clients will have different rendering and compute power and available memory and disk storage; paths to different clients will have different bandwidth and latency; server loads and hence the number of available server cycles will vary.

There a number of implications: (1) Clients must contribute significant compute resources; (2) Services must be able to aggregate the compute resources made available by the individual clients; (3) Services must be able to adapt to different resources. The bottom line is that no single server design is appropriate for all conditions. Heavyweight services must be *performance-portable*, adapting automatically to different levels of available resources.

In this paper we describe a run-time system and its API for building heavyweight network services. The system is based on the notion of *active frames*, which are mobile code and data objects that hop from host to host in a computational grid [4] (Section 2). Adaptivity is achieved by allowing application-level scheduling [3] of the work performed at each hop and the location of the next hop (Section 3). The active frame system is the basis for a remote visualization application (Section 4). In section 5 we evaluate the active frame system as a mechanism to aggregate resources in two different setups.

2 Active Frames

An *active frame* is an application-level transfer unit that contains program and associated application data. Frames are processed by compute servers called *frame servers*. The frames are transfered among servers using TCP/IP over a best-effort internet. The program and data contained in the frame are read by servers, which interpret and execute the program (See figure: 1). The active frame interface declares a single `execute` method, which is used as the entry point for the frame execution.

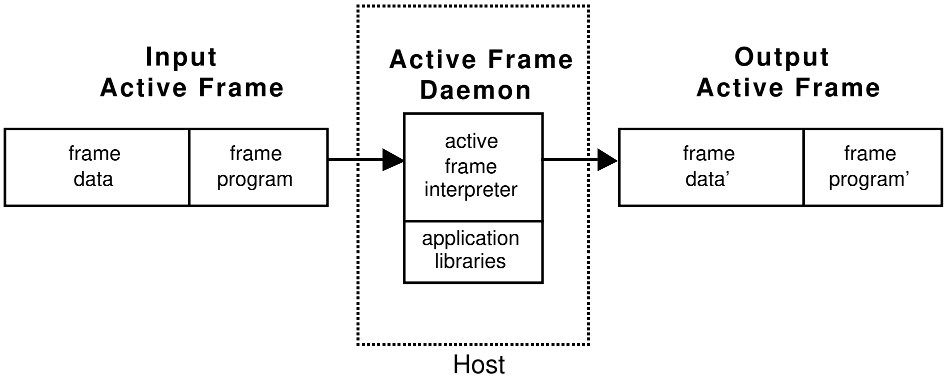

Fig. 1. Active frame and server

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

The `execute` method operates on the input data and produces output data. The frame has access to the state of the server where it is executing through the `state` parameter passed to the execute method. After the frame execution the server sends the frame to the address returned by the `execute` method.

2.1 Frame server API

The frame server defines a small set of primitives shown in Table 1. The application running on a host can either start a new server on its local host or access an already

Method	Description
void send(ActiveFrame frame)	Execute and send an active frame to the server returned by its execute method.
Object get(String id)	Retrieve an object from the server soft storage
void put(String id)	Add an object to the server soft storage

Table 1. Frame server API

running server on the same machine. The application then creates a frame and uses the `send` primitive to initiate the frame's processing. The `send` primitive on the server calls the frame's `execute` method and sends the frame to the returned address if it is not null. 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). An active frame executing at a server calls these primitives through the reference to the server's state passed to the `execute` method. Here is a sample frame that uses soft storage to keep state between execution of frames:

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

The frame uses the `get` method (line 4) to obtain a previously stored estimate of the time it will take to complete a computation. After the frame finishes the computation (line 5) it updates and stores the estimate in the server's soft storage using the `put` method (lines 6-7).

With this simple mechanism the application can exploit the compute resources of the frame servers by shipping part of the computation along with the data to the servers. In order to provide more sophisticated services to the application, the servers can be extended with application specific libraries. This is a crucial feature because it allows the use of existing packages and integration with other systems. For example a visualization program can use packages like vtk [7] or openDX [1]. In our current implementation the servers load these libraries at startup time and active frames can access the libraries throughout their execution.

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

3 Scheduling with active frames

The scheduler interface defines a single method:

```
interface Scheduler {
  HostAddress getHost(int hopCount);
}
```

The `getHost` method returns the address of the next host the frame should be sent to. It takes a single integer argument that specifies the number of servers the frame has visited. A zero argument indicates that the frame is at the *source server*.

The scheduler is transmitted along with the frame program. The `getHost` method is called by every server the frame visits. This mechanism permits the implementation of various scheduling policies, allowing the application to make decisions at different points during its execution. Here are some examples.

Static scheduling: 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 application can exploit information about its own resource demands to create the schedule. However it cannot adapt to dynamic resource availability. The following example shows the implementation of a static scheduler:

```
class StaticScheduler
    implements Scheduler {
 StaticScheduler(HostAddress[] list) {
  this.hosts = list;
 }
 HostAddress getHost(int hopCount) {
  if(hopCount<numberOfHosts) {
   return this.hosts[hopCount];
  }
  return null;
 }
}
...
HostAddress[] list = getHostList();
Scheduler sched    = new StaticScheduler(list);
ActiveFrame frame  = new SampleFrame(sched);
server.send(frame);
```

The application computes the list of hosts to execute once 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 frame creation time: In this scenario, the frames are scheduled at the source server. The scheduler can combine application-specific information with data from a resource monitoring system such as Remos [5] or NWS [8] to make scheduling decisions. This scheme adds some 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 (`hopCount==0`) and uses that path for the frame transmission.

```
class AtCreationTimeScheduler
    implements Scheduler {
 ...
 HostAddress getHost(int hopCount) {
  if(hopCount==0) {
    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;
  }
}
```

4 Motivating remote viz service

We have the active frame system to implement a general remote visualization service called Dv [1]. The specific viz application (Quakeviz) produces an animation of the ground motion during the 1994 Northridge earthquake [2].

Each Quakeviz input dataset consists of a 3D tetrahedral mesh and a sequence of frames, where each frame describes the magnitude of the earth's displacement at each node in the mesh at a particular point in time. Ranging in size from hundreds of GB to TBs, the datasets are generally too massive to copy from the remote supercomputer site where they are computed to our local site. We must manipulate them remotely, and thus the need for a remote viz service.

The Quakeviz input datasets for the examples in this paper were produced by our `183.equake` benchmark program in the SPEC CPU2000 benchmark suite [6]. The visualization of the `183.equake` dataset produces a 3D animation of the wave propagation during the first 40 seconds of a quake.

Figure 2 shows the transformations that are applied to each frame in the dataset to produce a frame in the animation. The first step selects a region of interest both in time and space from the dataset after it is loaded from a file. The next step finds regions in the dataset with similar features to generate isosurfaces. The final tasks apply a color map to the values in the dataset, synthesize a scene, and then render the final image.

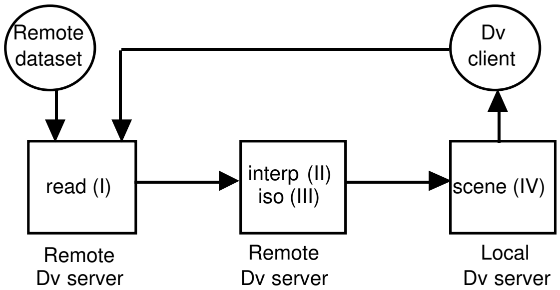

Fig. 2. A QuakeViz application implemented with Dv

4.1 Dv

Dv is a toolkit built on top of the active frame system that enables a user to visualize and interact with datasets stored at remote sites [1]. Figure 2 shows an example Quakeviz application implemented with Dv. This particular instance of the visualization runs on three hosts: (1) a *source host* that reads the dataset; (2) a *compute host* where the isosurface extraction is performed and a client host that that performs isosurface extraction; and (3) a *client host* that performs the scene synthesis and rendering. Each Dv server is a frame server that is extended with existing visualization libraries. The Dv client sends a special type of active frame, called a *request frame*, to the source host. The request server reads the dataset and generates a series of *response frames* that are sent back to the client. Examples of these frames are shown below.

```
class RequestFrame
  implements ActiveFrame {
 HostAddress execute(ServerState state) {
  Source source = new QvDataSource();
  ...
  while (moreFrames()) {
   Dataset data
```

```
7:      = source.getFrame(roi_parameters);
8:   ResponseFrame fr=new ResponseFrame();
9:   fr.setData(data);
10:  Scheduler sched = createScheduler();
11:  fr.setScheduler(sched);
12:  state.getServer().send(fr);
    }
    ...
13: return null;
   }
  }
```

The request frame creates a source object (lines 4–7), which is defined in the visualization library, to access the dataset. Then it creates a response frame (line 8) and a scheduler (line 10). The request frame sends the response with its scheduler (lines 11-12) using the frame server API. The request frame returns null (line 13) after it has sent the response frames back to the client.

```
1:class ResponseFrame
2:    implements ActiveFrame {
   ...
3:  HostAddress execute(ServerState state) {
      ...
4:    Filter filter = new Filter();
5:    filter.setInput(this.data);
6:    filter.compute();
7:    this.data = filter.getOutput();
      ...
8:    return scheduler.getHost(hopCount);
    }
  }
```

The response frame uses a filter from the visualization library to process the dataset (lines 4-6), and then it updates the frame data with filter output (line 7). Finally the response frame calls the scheduler to obtain the address of the next host and returns it.

5 Evaluation

It is clear that the active frame system provides a flexible mechanism for application-level scheduling. In this section we begin to evaluate the costs and benefits of this flexibility.

5.1 Single-host setup

This section characterizes the execution time on a single host of the visualization described in Section 4, using a relatively small dataset as input. The input dataset contains

`183.equake` 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. 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

Table 2. Mean execution time (local)

Table 2 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 3 and 4 show the execution time of isosurface extraction and rendering. Execution time for each tasks depends on the *number-of-isosurfaces* parameter (See table 2). 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 4 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.

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 Table 2) 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.

5.2 Pipeline setup

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

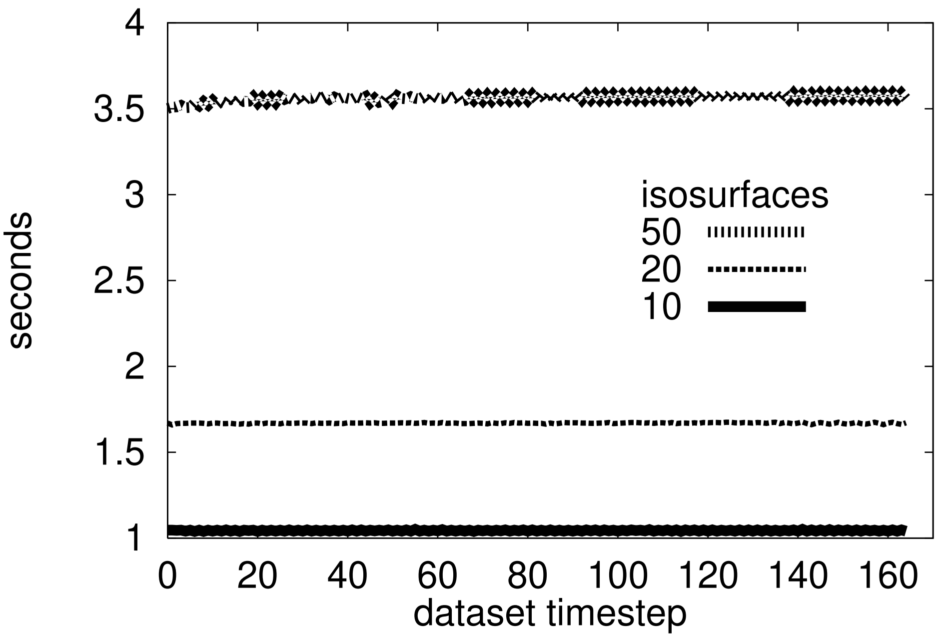

Fig. 3. Isosurface extraction (local).

hosts are used as compute servers and the other as a display client. The client host is the same as in Section 5.1. 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 Mbit switched Ethernet. The client communicates with the servers through a 10Mbit Ethernet.

The scheduler used in this configuration assigns the tasks to the three hosts as follows (See figure 5): 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.

Table 3 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 table 4) 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 for the extraction of 50 isosurfaces because the polygonal structure is much more detailed and complex in this case, requiring more bandwidth.

Despite the additional cost of transferring the data and the program, it is possible to obtain the benefits of resource aggregation. Table 4 shows the time each host spends processing a single frame and the respective frame rate each host can produce. For all

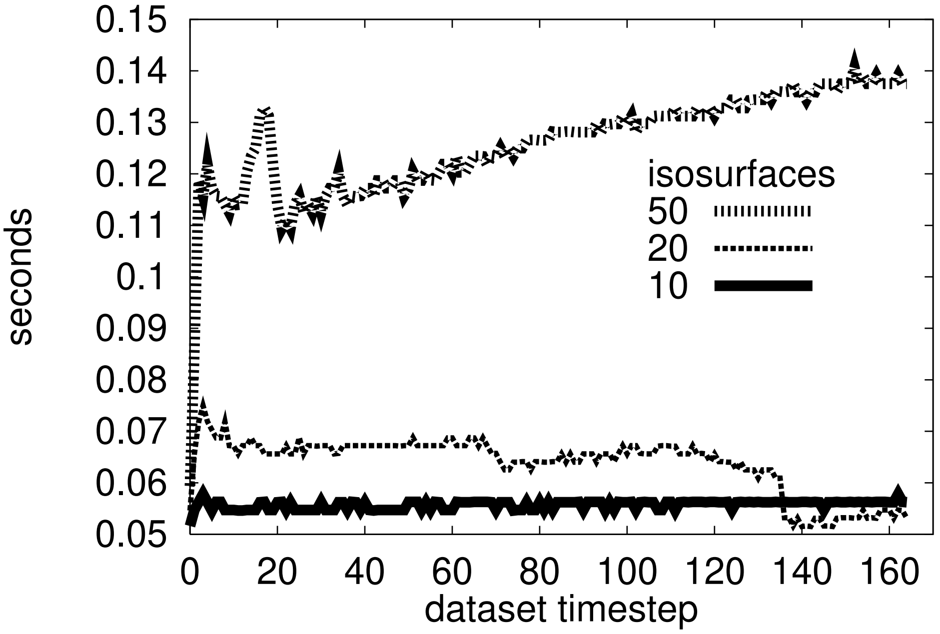

Fig. 4. Rendering (local).

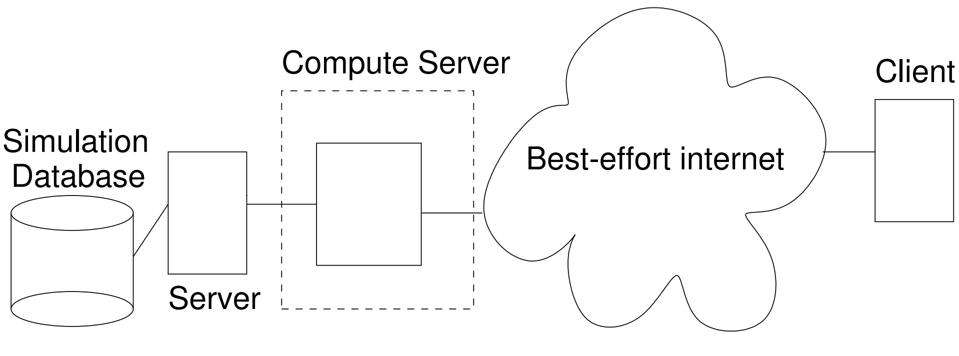

Fig. 5. Pipeline setup

# 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

Table 3. Mean elapsed time (pipeline)

	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

Table 4. Mean frame rate (pipeline)

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 6 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.

5.3 Fan setup

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 7). The frames are collected, reordered if necessary, and rendered by the client.

Table 5 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 Table 6). However, 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.

The rendering time is increased even further (See Table 6) and exhibits higher variability (See Figure 8), because the client host is under high load, which causes the rendering task to be preempted more often.

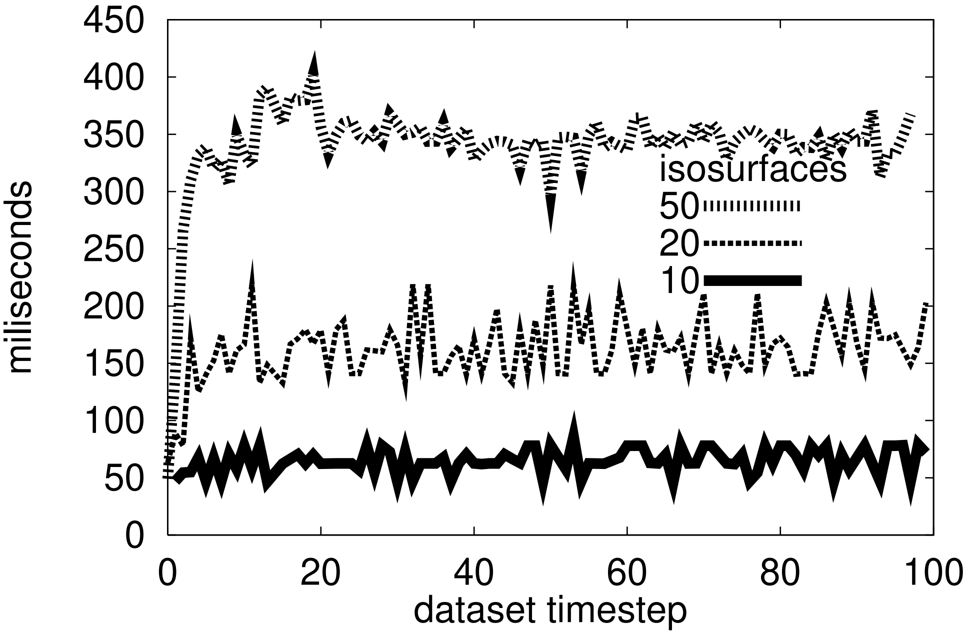

Fig. 6. Render (pipeline).

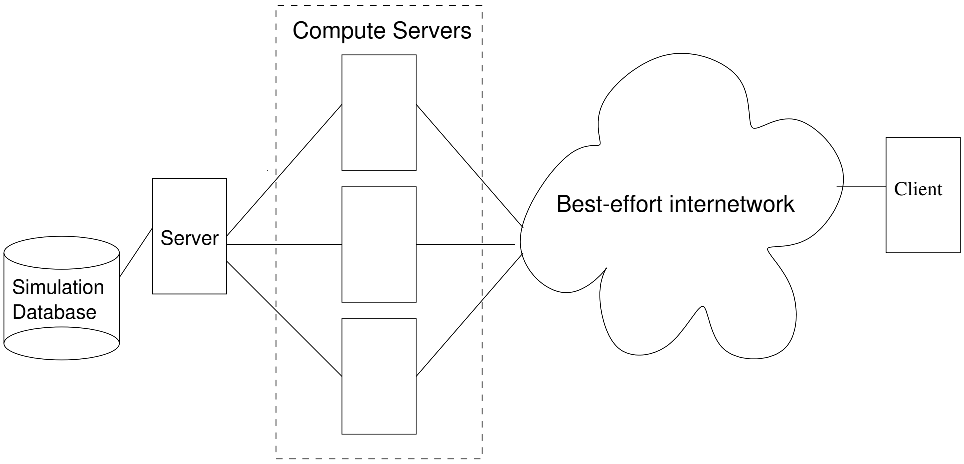

Fig. 7. Fan setup

# 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

Table 5. Mean frame rate (fan)

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

Table 6. Mean execution time (fan)

6 Conclusions

In order to provide heavyweight services in the future, we must build them so that they are performance-portable, in the sense that they can adapt to heterogeneous and dynamically changing resources. The approach we described here, based on the active frame mechanism, supports various levels of adaptivity, including completely static scheduling, scheduling at request time, and scheduling at frame delivery time.

Our evaluation suggests that remote viz is an example of a heavyweight service that benefits from this adaptivity. We evaluated the remote viz service under two different resource configurations. Although the flexibility of the mechanism introduces non-negligible overhead, in all cases the service obtains the benefits of resource aggregation.

References

1. AESCHLIMANN, M., DINDA, P., KALLIVOKAS, L., LOPEZ, J., LOWEKAMP, B., AND O'HALLARON, D. 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)* (Las Vegas, NV, June 1999), pp. 1833–1839.
2. BAO, H., BIELAK, J., GHATTAS, O., KALLIVOKAS, L., O'HALLARON, D., SHEWCHUK, J., AND XU, J. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering 152* (Jan. 1998), 85–102.
3. BERMAN, F., AND WOLSKI, R. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96* (August 1996), pp. 100–111.

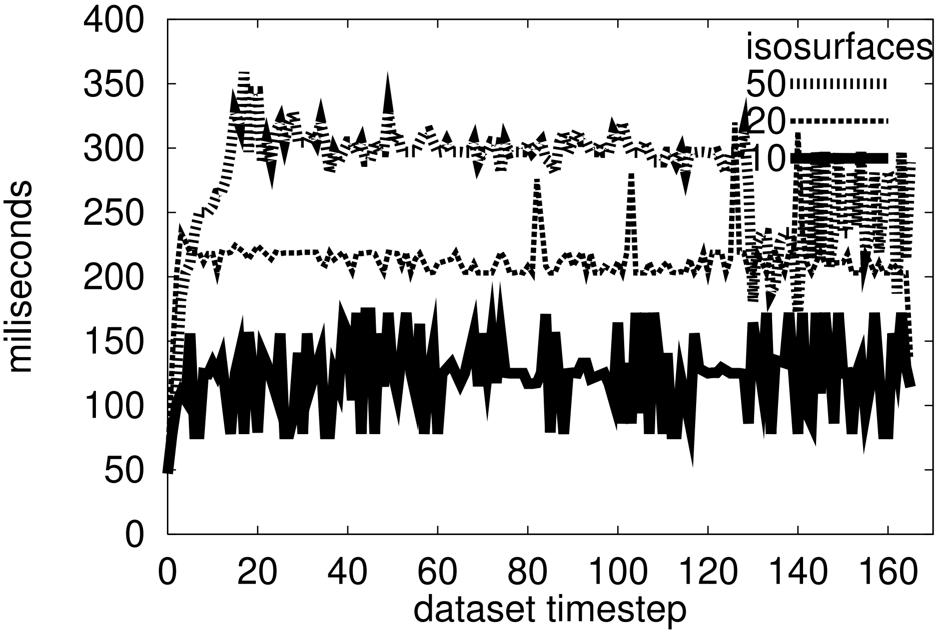

Fig. 8. Render (fan).

4. FOSTER, I., AND KESSELMAN, C., Eds. *The Grid: Blueprint for a New Computating Infrastructure*. Morgan Kaufman, 340 Pine Street, Sixth floor, San Francisco, CA 94104-3205, 1999.
5. LOWEKAMP, B., MILLER, N., SUTHERLAND, D., GROSS, T., STEENKISTE, P., AND SUBHLOK, J. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symp. High-Performance Distr. Comp.* (jul 1998).
6. O'HALLARON, D. R., AND KALLIVOKAS, L. F. The SPEC CPU2000 183.equake benchmark. www.spec.org/osg/cpu2000/CFP2000/, 2000.
7. SCHROEDER, W., MARTIN, K., AND LORENSEN, B. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, second ed. Prentice Hall PTR, Upper Saddle River, NJ, 1998.
8. WOLSKI, R. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference (HPDC97)* (Aug. 1997), pp. 316–325. extended version available as UCSD Technical Report TR-CS96-494.