Evaluation of a Resource Selection Mechanism for Complex Network Services

Julio C. López David R. O'Hallaron

Carnegie Mellon University
Pittsburgh, PA 15213, USA
{jclopez,droh}@cs.cmu.edu

Abstract

Providing complex (resource-intensive) network services is challenging because the resources they need and the resources that are available can vary significantly from request to request. To address this issue, we have proposed a flexible mechanism, called active frames*, that provides a basis for selecting a set of available distributed computing resources, and then mapping tasks onto those resources. As a proof of concept, we have used active frames to build a remote visualization service, called* Dv*, that allows users to visualize the contents of scientific datasets stored at remote locations. In this paper, we evaluate the performance of active frames, in the context of Dv. In particular, we address the following two questions: (1) What performance penalty do we pay for the flexibility of the active frames mechanism? (2) Can the throughput of a service based on active frames be predicted with reasonable accuracy from micro-benchmarks? The results of the evaluation suggest that the overhead imposed by active frames is reasonable (roughly 5%), and that simple models based on micro-benchmarks can conservatively predict measured throughput with reasonable accuracy (at most 20%).*

1 Introduction

In a typical client-server transaction, the server performs a fast and simple operation, such as fetching a file, querying a database, or running a small script, in response to a client's request. Service providers are generally unwilling to support complex (i.e. resource-intensive) services because of the potential for unacceptably high loads on their systems. Thus the following general question: How might we provide complex network services without overly stressing the resources of the service providers?

To be more concrete, suppose we would like to implement a complex network service that allows our colleagues to interactively visualize a large scientific dataset stored at our site. To access this service, a user requests some chunk of time and space (the region of interest) to be viewed as some number of isosurfaces at a particular resolution and from a particular distance and angle. The response is a complex sequence of tasks: reading the appropriate floating point numbers from the dataset, downsampling to the desired spatial and temporal resolution, converting the numbers to polygonal isosurfaces, rendering the polygons to pixels, and then painting the pixels on the user's screen.

So how might we implement this kind of complex service? One extreme is to perform all the computation on the server host. While this approach is good for clients, it might place unacceptably high load on the server. The other extreme is to perform all of the work on the client host. While this approach reduces load on the server, in some cases it requires higher bandwidth to transfer the data to the client and the client host might not have sufficient resources for the job. For example, the user may be accessing the service from a laptop or PDA with limited memory and graphics capabilities. As a compromise between these two extremes, we may need to partition the work between the client and the server hosts. And in some cases, when the combined resources of the client and origin server are insufficient, we may need to employ additional compute hosts, partitioning the work among the origin server, the compute hosts, and the client host.

The point is that different requests require different resources and different mappings of tasks to these resources. To have any hope of providing complex resource-intensive network services, two issues must be addressed:

1. Flexible and efficient mechanisms for selecting and aggregating hosts (and the networks that connect them), and for assigning and running tasks on these hosts.

2. Resource-aware application-level scheduling algorithms[4] to guide the mechanism in (1).

To address (1), we have developed a flexible mechanism

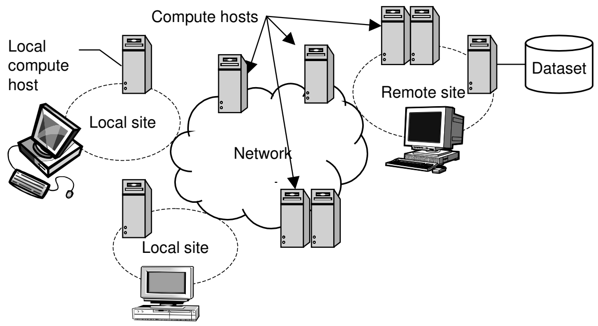

Figure 1. Scenario for a service based on active frames.

based on Java [12], called *active frames*, that allows services to easily move computations and data among different hosts. We have used the active frames mechanism to build a remote visualization service, called *Dv*, also based on Java, that allows us to partition a sequential visualization program on arbitrary configurations of distributed hosts [14, 15]. Figure 1 summarizes the basic idea, where a remote visualization service uses remote hosts storing the dataset, other compute hosts at the remote site, hosts at intermediate points in the network, as well as hosts at the various local sites. Active frames and Dv are described in Sections 2 and 3.

This paper addresses the following two questions about active frames, in the context of a remote visualization service:

- *What performance penalty on the hosts do we pay for the flexibility of the active frames mechanism?* This is an important question because the decision to base active frames on Java raises the specter of unacceptably high host overheads. The results of the micro-benchmarks in Section 4.1 indicate that the active frame host overheads are reasonable, on the order of 5% of total running time (the remainder is spent executing visualization routines written in native C++ code).

- *Can the frame rates of a service based on active frames be predicted with reasonable accuracy from micro-benchmarks?* This is an important question because it is a sufficient condition for designing effective application-level scheduling algorithms. If we can estimate the frame rates of different resource configurations, then we have a basis for choosing one configuration over another. It is also a non-obvious question because of the uncertainties introduced by the Java thread scheduling and garbage collection mechanisms. The results in Sections 4.2–4.4 give us some confidence that throughput is indeed predictable. Since the tests were run on a LAN with little cross-traffic, this claim is valid only for those cases where throughput is dominated by host performance rather than network performance. Given the complexity of tasks such as isosurface extraction, this claim is likely to hold for WANs as well, but a convincing answer must come from real tests.

2 Active frames

An active frame is an application-level transfer unit that contains both application data and a program. Figure 2 illustrates the basic architecture. Frame servers are user-level

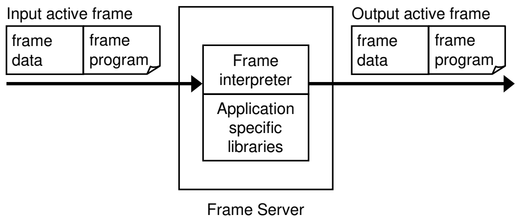

Figure 2. Active frame server.

Java processes that execute and forward active frames. The frame server consists of a frame interpreter and libraries with application-specific routines that can be called by active frames. The active frame interface declares a single `execute` method.

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

The frame program specifies the computation to perform on the data. Application-specific frames provide the appropriate implementation of the `execute` method. The `execute` method is called when the frame arrives at the server. When the frame execution finishes, the frame server forwards the frame to the server with 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. Resource intensive services can move computation and data to take advantage of the extra resources other compute servers offer. In essence, a service can use application- specific information and request parameters to select resources to satisfy a request.

Frame servers do not provide many services besides executing and forwarding frames. Instead, frame servers can be extended using Java's dynamic class loading or through application-specific libraries. The dynamic code loading mechanism is well suited for small programs, like "glue code", carried by the active frames. The application libraries are well suited for complex or compute intensive code or when size of the code makes it impractical to load it on demand over a slow network. The application libraries are co-located with the servers. The frame servers load these libraries at initialization time. Later, application-specific active frames can call into these libraries through JNI [11, 7], which is Java's mechanism to execute platform dependent code. This mechanism also has the advantage that it permits the use of existing code and libraries with active frames.

3 The Dv remote visualization service

Dv is a remote visualization service built on top of active frames. The service is provided using a series of frame servers specialized with vtk [22] libraries and C/C++ routines for transferring datasets over TCP connections. Dv enables users to visualize datasets stored at remote hosts.

The Dv service can be thought of as a series of queries to a remote dataset. The data is transformed by a series of filters to produce a visual representation of the data. Figure 3 shows the basic idea. The Dv client sends an active frame with a request to the server with the dataset. The server sends back to the client a series of active frames with the response(s). These response frames are executed by the frame servers in the path back to the client, transforming the data. Transformations are expressed in the form of a *visualization flowgraph*, where each node i denotes a filter and each edge (i, j) denotes that the output of filter i is the input of filter j. The nodes of the flowgraph are partitioned dynamically by the active frames. For example, the first frame server might execute node 1, the second frame server nodes 2 and 3, and so on. This flexible partitioning scheme allows us to run the service under different resource configurations *without having to modify the application code.*

The Dv client is a frame server extended with an interface for reading user input and displaying the output. Dv defines various active frames used in the visualization service. To initiate a Dv session, the Dv client sends a special kind of active frame, called a *request handler*, to a server, called the *source server*, that is co-located with the remote dataset. During a *session*, the Dv client sends a series of *request frames* to that Dv server. Each request frame contains visualization parameters, the flowgraph, and a scheduler that assigns flowgraph nodes to servers. The request handler produces a sequence of one or more *response frames*, which eventually end up back at the Dv client, where they are displayed.

Resource selection: Since nodes in the flowgraph can be executed independently we can use active frames to execute flowgraph nodes at different hosts. The application provides an implementation of the scheduler interface. The scheduler is called to obtain the mapping of flowgraph nodes to compute hosts. The scheduler can use application-specific information to decide where to execute a particular node of the flowgraph. The scheduler is called in every server the frame travels through to obtain the host where the next node in the flowgraph should be executed. This allows the implementation of dynamic resource selection policies at frame delivery time.

4 Evaluation

The evaluation answers the following two questions:

- *What is the overhead introduced by the active frame mechanism?* To answer this question, we use micro-benchmarks to characterize the elapsed times of individual visualization operations.
- *Can the frame rate observed in different resource configurations be estimated from measurements of elapsed times of individual operations?* To answer this question, we run the service using three different resource configurations (*base*, *pipe*, and *fan*) and we use the results from the micro-benchmarks along with simple models to estimate the frame rate of each resource configuration. Then we compare the estimated frame rate with the measured frame rate.

The input dataset was produced by the `183.equake` program from the SPEC CPU2000 benchmark suite [8, 20]. The dataset characterizes the simulated motion of the ground in a 50km x 50km x 10km chunk of the San Fernando Basin during an earthquake [3]. The volume of earth is discretized into an unstructured three-dimensional mesh with 30K nodes and 151K tetrahedral elements. The dataset contains 101 frames, one frame per output time-step. Each frame is 120KB, and consists of scalar values that correspond to the horizontal displacements of the mesh nodes. The visualization application generates an animation of all the input frames over the entire volume of the basin, computing various isosurfaces that track the advancing front of the seismic wave. For this evaluation we chose to compute 1, 5, 10, 15 and 20 isosurfaces.

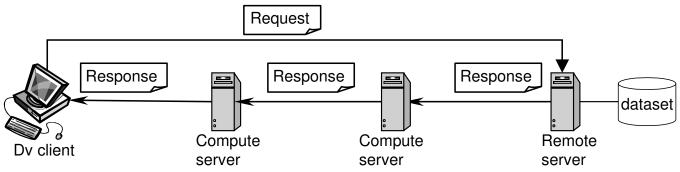

Figure 3. Dv execution model.

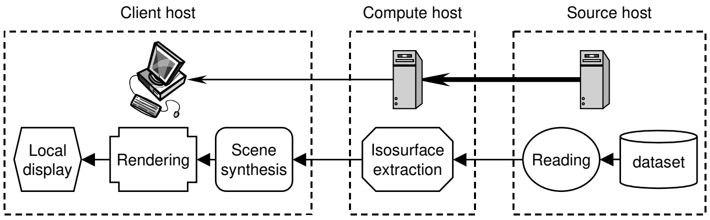

Figure 4. Micro-benchmark and pipe setup.

4.1 Micro-benchmarks

To determine the overhead introduced by the active frame mechanism, we measured the individual times of each step in the visualization, comparing the running times of the steps related to active frames to the total running time. This experiment involves the three hosts shown in Figure 4: a source host, a compute host, and a client host. The client host is a Pentium-III/450MHz machine with 512 MB of memory running NT 4.0 with a Real3D Starfighter AGP/8MB video accelerator. The compute and source hosts are 550 MHz Pentium III machines with 256 MB of memory and Ultra SCSI hard disks running version 2.2.16 of the Linux kernel. The compute and source hosts are connected with a 100 Mbps switched Ethernet. The client communicates with the servers through a 10Mbps Ethernet. The Dv servers run with JDK version 1.2.006 [17] using the classic Java VM [12] and no just-in-time compiler. The visualization library used to process the datasets is `vtk` version 2.4 compiled with `gcc`. The initial heap size for the frame servers was 128 MB and the maximum was 220 MB.

Executing the flowgraph using active frames requires the following steps (steps related to active frame processing are denoted with a "*"):

- *Read*: The read operation is performed by the source host. The dataset structure (the mesh) is loaded once at the beginning of a client session. This operation reads the time-step data (displacement values) to satisfy a particular request.
- *Mesh data transfer*: In this step the mesh data is transfered from the source host to the compute host.
- **Active frame transfer 1*: In this operation the state used in the active frame is transfered from the source host to the intermediate compute host. This includes marshalling, transferring, and unmarshalling the frame. This operation is directly related to the active frames mechanism.
- *Isosurface extraction*: This operation computes an isosurface using the values in the dataset. This is a CPU intensive operation.
- *Poly data transfer*: This step transfers the polygonal data generated by the isosurface extraction from the compute host to the client.
- **Active frame transfer 2*: In this operation the state used in the active frame is transfered from the compute host to the client. This operation is directly related to the active frames mechanism.
- *Render*: This operation synthesizes the scene, and renders and displays the output of the visualization process at the client.

For this experiment, the client requests only one visualization frame at a time. At any given point during the experiment there is only one frame in execution in the system. Each operation is executed sequentially so there is no interference from other operations or frames.

Figure 5 shows the mean time in milliseconds to complete each operation when only one isosurface is extracted. The time shown in the table is the average time over 50 runs of the visualization service. Each run processes 101 data frames (simulation time steps). The third column in Figure 5 shows the percentage of the total time taken by each operation.

Figure 6 shows the mean time in milliseconds per operation for various values of the number of isosurfaces parameter. As the number of isosurfaces increases, the computation

Operation	time (ms)	%
Read	269	16
Mesh transfer	555	32
AF transfer 1	38	2
Iso extraction	744	43
Poly transfer	29	2
AF transfer 2	55	3
Render	30	2
Total	1719	

Figure 5. Mean elapsed time per operation for 1 isosurface (micro-benchmarks).

needed to process a frame increases. The main contributor to the increase in the processing time per frame is the isosurface extraction operation itself. The poly data transfer and render times also increase due to the increase in the size of the isosurface extraction's output (more polygons to transfer and render). However, some of the operations in the process, including the ones related to active frames, are not affected by this parameter and their execution time does not change. The error bars in Figure 6 correspond to 90%

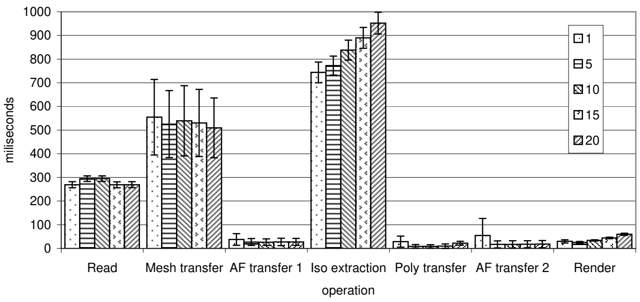

Figure 6. Mean elapsed time per operation (micro-benchmarks).

confidence intervals. The high variability of some of the operations can be explained. Figure 7 shows the elapsed time to perform the *isosurface extraction* step. The operation is very predictable until around the 35th time step is requested from the dataset. At this point, even though the average time to complete the operation is similar, a higher variability is observed. This behavior is an artifact of the implementation. When the frame server in the intermediate compute host consumes all the memory available in its heap, the garbage collector executes, halting the processing of frames. This affects all operations running on the compute server. The operations in Figure 6 with relatively high variation have a behavior similar to the one shown in Figure 7.

To summarize, the steps related to active frames account

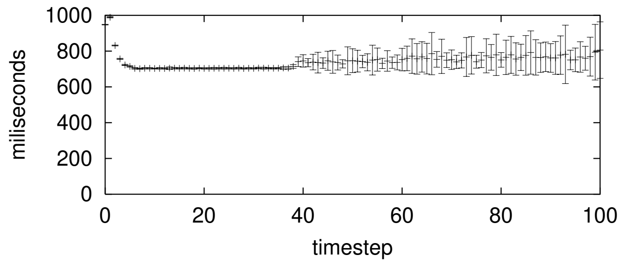

Figure 7. Isosurface extraction time (micro-benchmark).

for less than 5% of the total running time, which seems reasonable. In general, the processing of active frames remains fairly constant independent of application parameters, thus we expect that for application parameters that require more computation (e.g., more operations in the flowgraph) and larger datasets, the relative cost of processing active frames will be even lower. Also, we expect the size of active frames to be relatively small for most applications, since active frames act as glue code that call into libraries loaded in the servers. Thus we expect the cost of processing and transferring active frames to be low for other applications as well.

4.2 Base setup

Our second question asks whether the results from the micro-benchmarks are useful for estimating the frame rate of a visualization session. To answer this question, we executed the service with three different resource configurations: *base*, *pipe*, and *fan* and various values for the number of isosurfaces parameter (1, 5, 10, 15, 20). Then we compared the measured frame rate with the frame rate derived from the micro-benchmarks. Despite initial concerns that the variability introduced by the frame server implementation might make it more difficult to estimate frame rates, we find that we are able to conservatively estimate them using simple "back-of-the-envelope" models.

In the micro-benchmark experiments, frames are processed sequentially and thus there is no overlapping in the processing of different frames. In the following setups (base, pipe and fan) multiple frames belonging to a client session are processed simultaneously, and thus there is overlapping between the various operations of the flowgraph.

In order to obtain an estimate of the frame rate we use the following formula:

$$E[\text{fr}] = (\max_{j=\text{servers}}(D_j * E[V_j]))^{-1} \quad (1)$$

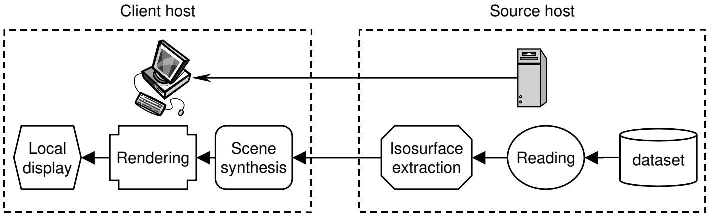

Figure 8. Base setup.

Variable D_j is the demand on server j and is given by:

$$D_j = \sum_{j=\text{ops}_j} (E[\text{op}_j])$$

$E[\text{op}_i]$ is the expected time to complete operation i of the flowgraph and obtained from the micro-benchmarks. D_j is the sum of the elapsed times of the operations that server j must execute in order to process a single frame. In essence, this is the time a server spends processing a single frame. $E[V_j]$ is the expected number of visits for a frame to server j (i.e., the fraction of the total number of frames that go through server j).

The base setup uses only the two hosts shown in Figure 8. The source host (right) stores the dataset. The source server executes the *read*, *isosurface extraction*, *poly data transfer*, and *active frame transfer 2* operations. Note that the *active frame transfer 1* and *mesh data transfer* operations are not necessary since *read* and *isosurface extraction* are executed in the same server. The client (left) executes the *poly data transfer*, *active frame transfer 2*, and *render* operations.

Host	D_i (ms)	$E[V_i]$	$E_i[\text{fr}]$ (fps)
Source	1096	1	0.91
Client	113	1	8.85

Figure 9. Demand on host per frame for 1 isosurface (Base)

Figure 9 shows the demand, expected number of visits and frame rate for each host when only one isosurface is visualized. According to the model the source server is the bottleneck and the frame rate is limited to 0.91 fps (frames per second). The observed frame rate for this setup is 1.04 fps when one isosurface is visualized (See Figure 10). In this case the estimate is within 12% of the observed throughput.

Figures 10 and 11 show the observed frame rate for the other values of the number of isosurfaces parameter. Each data point is the average across 50 runs. The observed frame rate is consistent despite the high variability observed in some operations in the micro-benchmark (See Figure 7).

# isos	Observed frame rate	Estimated frame rate	Error %
1	1.04	0.91	12
5	0.96	0.92	4
10	0.91	0.86	4
15	0.86	0.84	2
20	0.81	0.79	2

Figure 10. Estimated and observed frame rates (Base).

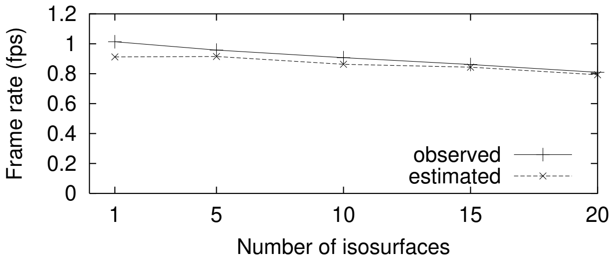

Figure 11. Frame rate (Base).

The estimate is lower because it assumes that while transferring a frame or application data, no other operation can be executed. This is clearly not the case since these operations are I/O bound, and can be interleaved with CPU bound operations such as isosurface extraction or rendering. Notice that as the ratio of compute operations/IO operations increases, the estimated frame rate approximates the observed frame rate.

4.3 Pipeline setup

This resource configuration is the same as the one used for the micro-benchmarks (See Figure 4). It involves three servers: the source server (right) executes the *read*, *mesh data transfer*, and *active frame transfer 1* operations; the compute server (middle) executes the following operations:

mesh data transfer, *active frame transfer 1*, *isosurface extraction*, *poly data transfer*, and *active frame transfer 2*. In contrast with the micro-benchmark experiment, multiple frames for a session are processed in the pipeline simultaneously to achieve a higher frame rate.

Host	D_i (ms)	$E[V_i]$	E_i[fr] (fps)
Source	862	1	1.16
Compute	1420	1	0.70
Client	113	1	8.85

Figure 12. Demand on host per frame for 1 isosurface (Pipe)

According to the model, this configuration has a lower frame rate than the base setup. The intermediate compute server becomes the bottleneck with a frame rate of 0.7 fps when extracting one isosurface (See Figure 12). The cost of transferring the data from the source server to the compute server offsets the benefit obtained by offloading the isosurface extraction from the source server to the compute server.

# isos	Observed frame rate	Estimated frame rate	Error %
1	0.83	0.70	15
5	0.83	0.74	11
10	0.79	0.70	11
15	0.76	0.68	11
20	0.72	0.65	10

Figure 13. Estimated and observed frame rates (Pipe).

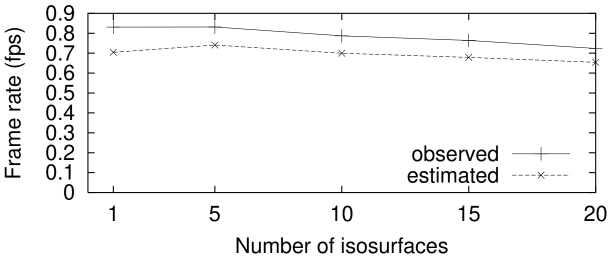

Figure 14. Frame rate (Pipe).

Figures 13 and 14 show the frame rate for different values of the number of isosurfaces parameter. Again the model produces a conservative estimate of the frame rate. The observed frame rate is 0.83 fps and the error for the estimation is 15% for 1 isosurface. (See Figure 13). The higher error in this setup, compared to the base setup, can be attributed to the additional IO operations needed in this setup, namely the mesh and active frame transfers.

4.4 Fan setup

The purpose of the fan configuration (Figure 15) is to increase the frame rate by reducing the load of the intermediate compute server. Each server now sees only $1/3$ of the total frames processed in a session.

Host	D_i (ms)	$E[V_i]$	E_i[fr] (fps)
Source	862	1	1.16
Compute	1420	1/3	2.11
Client	113	1	8.85

Figure 16. Demand on hosts per frame for 1 isosurface (Fan)

According to the model, in this configuration the source server becomes the bottleneck again. Figure 16 shows the expected demand, visits and frame rate for the visualization of one isosurface in the fan setup. In this case the source server limits the frame rate to 1.16 fps. The observed frame rate of 1.46 fps is consistent with the estimated frame rate.

# isos	Observed frame rate	Estimated frame rate	Error %
1	1.46	1.16	20
5	1.45	1.18	19
10	1.45	1.16	20
15	1.45	1.21	17
20	1.45	1.24	14

Figure 17. Estimated and observed frame rates (Fan).

Following the same procedure described above, we obtain the estimated frame rates, shown in Figures 17 and 18, for other values of the number of isosurfaces parameter.

Again the estimate is conservative (See Figure 18), in this case falling up to 20% short, relative to the observed frame rate. This is due to the increase in the overlapping between I/O and CPU operations, which the model does not take into account.

4.5 Evaluation summary

To summarize the results of the evaluation:

- The results from the micro-benchmark experiment show that the overhead introduced by the active frames mechanism is relatively low.

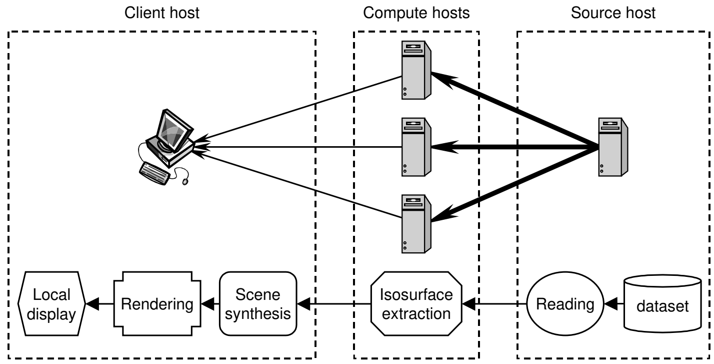

Figure 15. Fan setup.

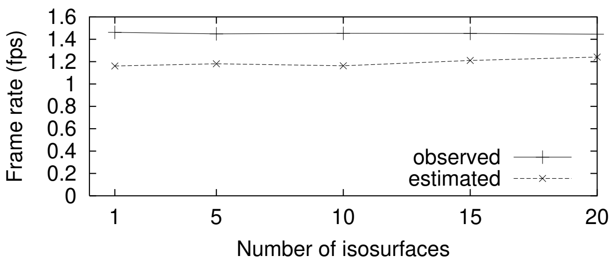

Figure 18. Frame rate (Fan).

- The results from the micro-benchmarks are also useful for predicting frame rates under three different resource configurations.

5 Related work

The use of application-level information to select resources was proposed by the AppLeS (Application Level Scheduler) work [4]. An AppLeS contains information about the structure of the application and its resource requirements. The AppLeS obtains information about resource availability from the Network Weather Service (NWS) and then partitions the application and selects the appropriate resources on behalf of the application. The work presented in this paper is inspired by the AppLeS work. We believe that in order to use the resources better and meet the application needs it is necessary to use information about the structure of the application and its resource needs.

Bundling programs with data has been explored in various contexts. Active Messages [24] are used in the communication messages of parallel programs. In Active Networks [23], capsules are used to perform limited "lightweight" computation in the network routers. This approach is used to deploy new protocols and support network administration tasks. The Active Network's transfer units, *capsules*, contain information about the "protocol handler" that should process the capsule. Active Services [1] proposes to push the "active" functionality towards the endpoints instead of placing this functionality in the network core. A similar approach is exploited in [6] to deploy multimedia services for resource limited clients. ABACUS [2] is used in System Area Networks with data intensive applications to support automatic function migration between the storage hosts and the clients. Our active frames approach provides a flexible mechanism to move data and code in resource intensive application.

Process level migration and application mobility is supported in systems like Condor [13], Sprite [5]. These systems try to support transparent process migration usually to take advantage of underutilized resources and balance the load in a cluster of workstations. Similarly systems like PUNCH [10] and NeOS [18] allow the remote execution of complex optimization solvers or resource intensive application through a web interface. In general, the application has little control of when and where it will execute. Emerald [9] supports fine-grained application controlled mobility. Active frames support coarser grained mobility, meaning that the executing code cannot be migrated at any arbitrary point, but only at the boundary of coarse level operations. This approach also gives more control to the application to decide where and when to execute a particular operation. Our framework relies on the mechanisms provided by Java [12] to move the computation among hosts.

Systems like REMOS [16] and NWS [25] provide information about resource availability on a computational grid. The information provided by these systems could be used to drive the resource selection for our visualization service and other services built on top of active frames.

Odyssey [19, 21] provides support for application adaptation in the context of mobile and ubiquitous computing. Odyssey introduces the concept of *fidelity* to define the quality of the output or service the user gets. A mobile device running Odyssey (i.e., a laptop) monitors the usage of resources like CPU, battery, network and cache state in the system. Odyssey provides feedback to the running applications to adjust their fidelity levels and thus the resource usage to meet a certain goal (i.e,: battery duration). We think Odyssey's approach and ours are complementary. For example, if a user accesses our remote visualization service from an Odyssey enabled device, the visualization service could obtain feedback from Odyssey and allocate resources accordingly to meet the user's goals.

6 Future work

So far, we have been able to run our remote visualization service in a LAN environment with relatively small datasets. The active frame mechanism has enabled us to manually choose from various resource configurations to run our visualization service without modifying the application.

Our long term goals include (1) automating the resource selection process and (2) enabling the visualization of larger datasets over wide area networks.

The basic steps of an automatic resource selection mechanism include resource requirements characterization, resource availability prediction/reservation and the resource selection itself. Our current system provides some basic capabilities to characterize the resource requirements (CPU and network) to execute a particular operation of the application, given some input parameters. In order to automate the resource selection process we need a robust mechanism to characterize these resource requirements. For example, how does the system provide resource requirement data when the application executes with a set of parameters that the system has not seen before. Our current system works under the assumption that resources are dedicated to a particular application. In order to enable complex network services like our remote visualization service over WANs, it is necessary to account for other users' requests and applications competing for resources. Dynamic information about resource availability from systems like REMOS and NWS is needed to make the appropriate resource selection. Finally, the resource selection mechanism has to be robust to avoid oscillations and at the same time it has to be lightweight, so as avoid a high overhead in the system. In essence an on-time suboptimal decision is more valuable that a late optimal decision.

7 Conclusions

The deployment of resource-intensive network services is challenging because of the dynamics of application resource consumption and resource availability. Often, the available resources are insufficient to satisfy a user request. In order to overcome these challenges, applications should be able to select and aggregate multiple resources using application-level information. We have presented a flexible mechanism, called active frames, that allows applications to easily move computation and data among compute hosts. We have shown that the overhead introduced by the active frames mechanism is reasonable. The active frame mechanism allows the use of application-level information to effectively take advantage of the resources at the compute hosts. We have used the interface provided by the active frames to allow the user to statically select the resources used to satisfy a request of our visualization service. We used simple models to characterize the expected frame rate of a particular resource configuration. The results suggest that it is possible to automate the process of resource selection using a process similar to the one we did manually to estimate the observed frame rate.

Acknowledgments

We gratefully acknowledge the contributions of Martin Aeschlimann, Peter Dinda, and Bruce Lowekamp in the development of the active frames and Dv systems. Loukas Kallivokas gave us helpful direction on the visualization process. The work was 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.

References

[1] E. Amir, S. McCanne, and R. Katz. An active service framework and its application in real-time multimedia transcoding. In *SIGCOMM'98*, pages 178–189. ACM, Sep 1998.

[2] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement in active storage clusters. Technical Report CMU-CS-99-140, Carnegie Mellon University, 1999.

[3] 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, Jan. 1998.

[4] F. Berman and R. 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.

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

[6] A. Fox, S. Gribble, E. Brewer, and E. 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, Oct. 1996. ACM.

[7] R. Gordon. *Essential JNI: Java Native Interface*. Prentice Hall, 1998.

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

[9] 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.

[10] N. Kapadia and J. Fortes. On the design of a demand-based network-computing system: The purdue university hubs. In *Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

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

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

[13] 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.

[14] J. Lopez and D. O'Hallaron. Runtime support for adaptive heavyweight services. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)*, volume 1915 of *Lecture Notes in Computer Science*, pages 221–234. Springer-Verlag, Rochester, NY, 2000.

[15] J. López and D. O'Hallaron. Support for interactive heavyweight services. Technical Report CMU-CS-01-104, Carnegie Mellon University, Pittsburgh, PA, February 2001. http://www.cs.cmu.edu/ dv.

[16] 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.

[17] S. Microsystems. Java SDK. http://java.sun.com/j2se.

[18] J. More, J. Czyzyk, and M. P. Mesnier. The NEOS server. *IEEE Journal on Computational Science and Engineering*, (5):68–75, 1998.

[19] B. Noble, M. Satyanarayanan, D. Narayanan, J. 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.

[20] D. R. O'Hallaron and L. F. Kallivokas. The SPEC CPU2000 `183.equake` benchmark, 2000. www.spec.org/osg/cpu2000/CFP2000/.

[21] 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.

[22] 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.

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

[24] 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.

[25] R. 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, Aug. 1997. extended version available as UCSD Technical Report TR-CS96-494.