Application Management Techniques for the Bifrost System

Avneesh Bhatnagar, Evan Speight
Cornell University
Computer Systems Lab
Ithaca, New York
{*avneesh,espeight*}*@csl.cornell.edu*

Dan Crawl, Joseph Dunn, John Bennett
University of Colorado
Department of Computer Science
Boulder, Colorado
{*crawl,josephd,jkb*}*@cs.colorado.edu*

Abstract

This paper describes the adaptive component migration facility in the Bifrost Location-Independent Computing System. We present a facility that allows mobile clients to selectively invoke application component functionality locally or remotely in order to improve user response time. The Bifrost runtime system uses a decision-making process that takes into account a variety of issues affecting the migration decision, including client and server resources, component size, the size of the data associated with the component, and network characteristics. We present a detailed design of this system, while examining the options of function call re-direction and API wrappers as a means to extend the semantics of the underlying remote execution technology (DCOM). The adaptive functionality provided by Bifrost resulted in a minimum 29% reduction in response time experienced by the client over a default DCOM-based implementation for mobile client devices.

1. Introduction

Data, applications, and devices present in current personal computing environments are fragmented into a complex and hard-to-manage collection of information tools. The variety of information representations used by these tools further hinders their effective inter-operation. The ability to incorporate higher computational power and high-quality, widely-connected communication into virtually any device offers the promise of a new generation of personal computing devices that can be easily personalized to adapt to individual needs of users. In order to ensure effective use of this proliferation of technology, the Bifrost Location-Independent Computing System seeks to provide an intuitive environment for users to access data and application functionality in a world that is almost-always connected.

Bifrost distinguishes itself from related research by merging the way in which applications and data are managed in a mobile system. Bifrost implements a new method for delivering application functionality to remote clients in the form of *application templates* that allow the necessary functionality to be "paged in" to a mobile client as necessary, based on adaptive runtime decisions. We use the concept of *affinity* to enable application and data management. Affinity represents the strength of attraction a particular application component or piece of data has for a device, user, or location. In this way, the Bifrost runtime system connects related data, people, and applications in order to facilitate the efficient movement of data in a resource-constrained mobile system. The concept of *affinity-directed mobility* allows the development of mechanisms and algorithms that Bifrost employs to adaptively decide where and when to move data. Finally, each Bifrost user is represented by an opaque object in the system known as a *persona*. This persona has an associated personal information region (PIR) that contains all data with affinity values and application usage characteristics for an individual or group persona. This paper focuses on the issues associated with application and data management in Bifrost.

Application Mobility

Mobile devices are currently constrained by fair-to-poor network connectivity and resource limitations such as disk space, memory, and processor speed. This results in the development and deployment of stripped-down applications on personal communication devices such as PDAs, accompanied by a loss of application functionality over their full-fledged desktop counterparts. Bifrost attempts to provide application management and full-application functionality in the face of these device constraints. We assume network connectivity in a similar form to that of 802.11 in a centralized/adhoc mode, or easy access to service providers through a wired medium. Other forms of connectivity (e.g., Bluetooth [7]) can also be easily taken into account. We use current implementations of data consistency mecha-

nisms [28] and assume that the client maintains a cached copy of data, which is reconciled periodically.

Since Bifrost is not a systems security-related project, we build upon the implementation of authentication and access control provided by the Windows operating system family [24]. The COM/DCOM runtime itself allows the system administrator to setup access control using standard API methods and a system-provided administration utility for COM servers (*dcomcnfg* [24]) built upon the Windows implementation of Kerberos [25]. The proposed Protected Extensible Authentication Protocol (PEAP) [26] provides a more scalable challenge/response mechanism for a 802.11 wireless LAN setup. Bifrost incorporates these features into a session-based authentication mechanism. The client is provided a unique session key each time they use the facilities provided by Bifrost.

Prior research (e.g., the Odyssey [32] system) has assumed that the application present on the device is capable of interpreting/executing the data that is accessed from the server. This assumption is typically violated on a small portable device, since it might not possess the necessary resources to interpret the data, or indeed to even hold the entire data set due to memory or disk limitations. We therefore divide an application into a "docked application" consisting of place-holders for application components to plug in as appropriate. For example, a text editor might contain GUI features entailing the use of a spell checker, image processing tool, etc., but the binary *components* responsible for this functionality might not be present on the device itself. Figure 1 presents a general overview of such a system. Client devices that are registered with Bifrost servers can interact with the servers when the user requests extended functionality. If the desired functionality is not locally available, migration decisions are made by Bifrost Service Agents to ensure best component placement.

In Bifrost, components may be "paged in" or "paged out" due to changes in the execution environment. Such decisions are based upon attributes such as device disk capacity, available memory, processor speed, size of the component to be migrated, and the associated data. Other parameters, such as power consumption, can be easily integrated into our migration decision-making process.

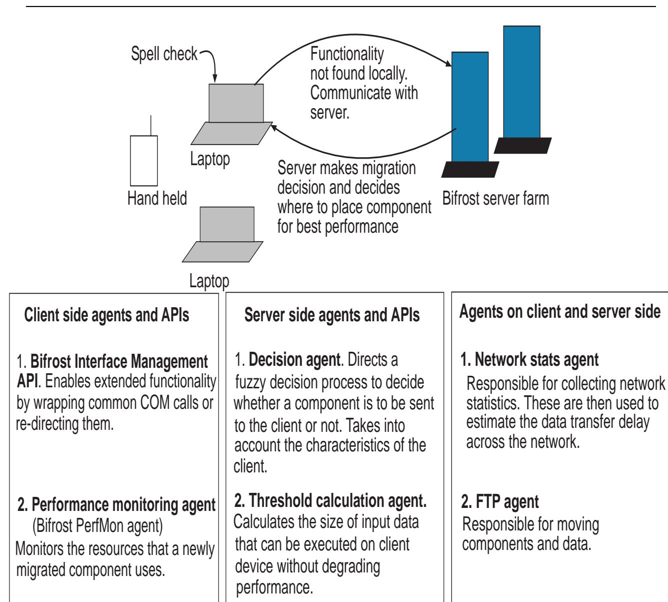


Figure 1. Overview of the system

2. Implementation

Bifrost is currently implemented on the Windows 2000 and Windows CE operating systems employing COM and DCOM [22, 40] to support distributed interoperation. COM (Component Object Model) and DCOM (Distributed COM) define a language-independent binary interface standard for packaging and distributing re-usable binaries, allowing components to expose polymorphic interfaces usable for interaction by client applications. The following section briefly explains how we have utilized COM/DCOM to provide the adaptive functionality [6]. We chose to use COM/DCOM in our Windows implementation because of the robustness of its implementation, ease of administration, and well-documented APIs.

2.1. Leveraging COM and DCOM

COM objects may be configured in several ways. As a DLL (dynamically link library), a COM object is referred to as an *In-Process* server since the DLL is loaded into the application's address space when COM invocation occurs. The advantage of in-proc servers is that bytes do not have to be transferred across different address spaces during an invocation, thus reducing overall latency. To increase scalability and facilitate distribution, a COM server can also be implemented as an *Out-of-Process* server, which is either an executable or a service launched upon system startup. These out-of-proc servers can be located on the same machine as applications, or at a remote server (in the case of DCOM). The advantage of this method lies in location independence for the executing component.

The Bifrost system seeks to adaptively combine the best of these two approaches by leveraging upon the DCOM infrastructure to create a "docked application template" as shown in Figure 2. In this figure, the server exposes some functionality for use by applications, but the executables (shown as dotted rectangles), might not be present on the client device. Instead, the empty "slots" mark place-holders where the real components fit if they are migrated to the device. We have configured each of these executables as a

COM component (server) that might be found on the client device or may exist at a remote location. We choose to migrate components between clients and servers as DLLs due to the lower latency of local invocation. To further promote re-use of components configured as DLLs, we employ surrogate processes [40] on the Bifrost server side to allow remote inter-operation.

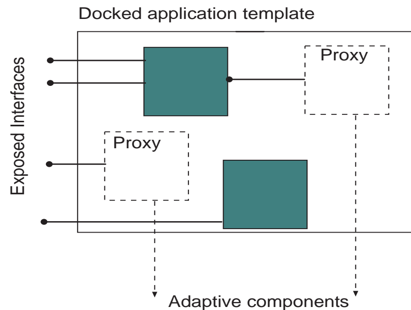


Figure 2. Docked application template

To guide the adaptive component decision-making process, Bifrost must have knowledge of when applications invoke COM and DCOM components. The execution location of the component is determined dynamically by the Bifrost runtime system based on runtime profiles and system characteristics, all of which are targeted toward providing the extended functionality that exists on full-fledged desktop applications to their mobile application counterparts. Bifrost establishes another layer between the actual component invocation call and the associated system call to enable application adaptability. We have investigated two different methods to provide this level of transparency to the user. We provide API wrappers to common COM methods, and we also implement function call re-direction for platforms that support it.

2.1.1. Transparent Invocation A common technique for tracking operating system calls is to place "hooks" into the source code statically or dynamically [34] to re-direct function calls of interest. The Detours [15] library, which is based upon this idea, intercepts Win32 functions by re-writing target function images. Using this library we re-direct common COM/DCOM calls to our own code, which then determines the best location from where the COM interface pointer is to be obtained. Table 1 shows the commonly-used methods that we re-direct. Similar re-direction can be done for the entire COM API [22].

In Bifrost, re-direction is supplemented with additional information (size of input data, calling process name, component name, etc.) when the component is invoked. These parameters help in the decision making process. If, for example, an invocation to a local component fails due to the component not being present on the device, these attributes are used to decide whether (a) the component can execute locally, and (b) whether the component should be migrated to the local device.

COM method	Functionality
CoCreateInstance()	Invokes a local COM component
CoCreateInstanceEx()	Provides remote invocation capability
CoInitialize()	Initializes COM library
CoUninitialize()	Closes COM library and frees resources

Table 1. Re-directed COM methods

The second approach toward providing transparency and adaptability involves wrapping common DCOM calls and providing the application writer a set of wrapper APIs (the *Bifrost Interface Management APIs*) that are used to create docked applications. We provide Bifrost wrapper functions for the same re-directed calls listed in Table 1.

Currently, only applications running on the x86 architecture can utilize the completely transparent function call re-direction mechanisms offered by Bifrost. Other system architectures (e.g., Windows CE) that do not support methods similar to *VirtualAllocEx()* [34], which allow a process to write into the virtual address space of another process (having been granted specific permissions), must use the provided Bifrost Interface Management APIs. Extending the support for re-direction is the subject of ongoing work.

3. Migration Phases

The Bifrost runtime system is responsible for determining the best execution site for a given component, taking into account factors such as network characteristics, data size, component size, and processing capabilities of the mobile device and the server. We have divided the migration decision process of the Bifrost runtime system into three design phases. The first phase represents the situation where a particular component has not been accessed before by a particular client. The only information available is the size of the component being requested and some characteristics of the requesting device. No performance data is available as yet, and therefore the decision to migrate a component is based solely on device capability. The second phase takes into account the performance differences of running a component on the client vs. executing the component on the server. The final phase incorporates network behavior and determines input data thresholds on the client for which the response time of executing the component is acceptable to the user.

3.1. Migration Decision Phase 1

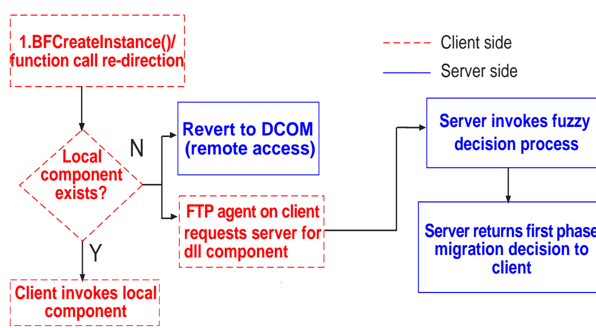


Figure 3. Sequence of events for phase 1

Figure 3 shows a summary of the events that occur in the first phase. The client calls the wrapper *BFCreateInstance()* (or the re-directed *CoCreateInstanceEx()*), which searches for a local component (in this case, a DLL). If this component does not exist locally, the remote Bifrost surrogate process is invoked to access the required functionality. Simultaneously, the FTP agent running on the client side requests the component from the server, where it exists in a component repository. Once the remote interface pointer is returned via DCOM, the decision process is started. The only decision that can be made at this point is to determine whether the device is *capable* of holding the component and the associated data, since no other execution information has been gathered.

We have approximated the capability of a mobile device using the following attributes: processor speed (P), amount of disk space available on the device (D), and the amount of available memory (M). To make an informed decision on whether to migrate a component, two additional attributes related to the application component are also helpful: component size (C) and the associated data size (Da). In the current implementation, the FTP agent uses a request descriptor containing the processor speed, amount of available disk space, and free memory to request the component functionality from the server. For future implementations we propose to expand this descriptor to include a unique identifier (generated using a time-stamp), the average processor utilization, the previous migration decision (for usage patterns as discussed in Section 6), the power consumption profile, and remaining battery power (building upon previous work [13, 41]). At low battery power, the tradeoff between the power requirements of the network necessary to run the component remotely and the processor power drain to run a resource-intensive algorithm locally is not immediately clear.

We have found processor speed to be a good heuristic for determining the device characteristics we are interested in. If a device has a slower processor, we do not expect it to have a large memory or disk configuration. However, no clear relationship exists between the above mentioned attributes to make a more informed decision. The best approach is to create a set of rules that are traversed to find the best match given a set of input values. For scalability, the resulting rule base should not experience an exponential increase in size due to addition of new attributes. At the same time, it can be agreed that some attributes would play a stronger role toward the final migration decision. Finally, there is a degree of uncertainty within this system since factors such as network degradation or processor utilization cannot be determined *a priori*.

Taking these considerations into account, the problem can be categorized within the class of decision-making problems under uncertainty. Such problems can be solved using a learning/feedback algorithm (which require a training phase), or by exploiting the conditional dependence relations between the attributes and creating joint probability distributions. The caveat is that these distributions might also become intractably large as the number of variables grows. In order to reduce the size of the rule-base and still utilize the probabilistic approach, the input values can be divided into categories based upon which the rule-base can be created. The *fuzzy set theory* [8] is based on this principle.

After several small experiments, we chose a subset of the fuzzy set theory, and, in particular, we utilized the mechanisms to categorize discrete input values. There are clear advantages to using fuzzy techniques for such a system [6, 10]. Since the resulting rule-base is independent of discrete input values, the system designer can accommodate faster and more capable devices without having to change the rule base, a significant advantage. The output values from the fuzzy rule system are: “Strong do not migrate”, “Undecided”, or “Strong migrate”, where relative execution times and network latencies help resolve the “Undecided” output. If the fuzzy engine returns “Do not migrate” as the migration decision for a component due to a large, associated auxiliary data set, we can accommodate cases where a smaller data set might provide a fairly respectable functionality. For example, a spell-checking tool may be shipped with a smaller dictionary of commonly used words.

3.2. Migration Decision Phase 2

The second phase of migration decision making extracts the performance information from the client device and uses this to decide whether remote or local execution will lower the user response time. During the first invocation (first phase), the interface pointer being used is accessed remotely (if a local component does not exist). The next invocation would yield a local pointer if the component has been migrated to the client device based upon the decision made in the previous phase. However, this may prove to be less effi-

cient than the remote execution because of significant processor resources consumed, and therefore increase the overall response time. To obtain this information we create a runtime profile of the execution of the component on the client machine.

3.2.1. Runtime Profiles Profile information from an application can be obtained statically or dynamically. Static techniques employ a source profiler, such as the PREP tool [27] in Windows. Beyond the obvious problems associated with the recompilation of the source code, this technique increases the size of the resulting binary even though profiling may not always be needed. The dynamic profiling approach employs hardware counters for tracking system resource usage. The Windows system library used for this purpose is *pdh.lib*, which provides API methods that are used to access raw CPU hardware performance counters and can be used to measure per process and per thread processor utilization. We employ a counter thread, which is part of the Bifrost Performance Agent, that wakes up periodically to collect these counter values.

Profiling is initiated in Bifrost when a component is migrated, runs for the first time, and subsequently exits. The post-execution analysis looks at the CPU time consumed by the component process, the overall process execution time, and the input data size. The goal is to improve the response time of the application as seen by the user, which is a relative measure that depends on how long the user is willing to wait for results to be obtained. Because of this, we also allow user-motivated statistics collection if the CPU utilization was high enough to impact the user's ability to interact with the device. For example, the user can stop the local execution of a slow image processing component and revert to remote execution of the component. The Bifrost Performance agent collects all such instances of user-motivated statistics or times when the process-related CPU utilization went above a certain threshold value. This information is sent to the Bifrost server process, where these statistics are compared with server-side statistics to gauge the relative performance. The next phase of component migration combines this data with current network quality to determine the best component location.

3.3. Migration Decision Phase 3

We now describe the final phase of the component migration decision, which involves setting up input data thresholds on the client device. These thresholds are subsequently used to quickly pick either a local or remote invocation of the desired functionality.

3.3.1. Collecting More Information During this phase, the Bifrost Server Agent extracts more information regarding the component execution on the client device. The application component is executed on the server, and similar profile information about CPU utilization and execution time is gathered for the particular data set. This information is then used to set up a threshold for the input data that can be safely run on the mobile client without degrading performance. The data is communicated to the performance monitoring agent on the client, which writes it to the system registry. At this point we have set up a sliding window for size of the input data. When a smaller size is encountered, the application can safely use the local interface pointer. If a higher value is encountered, the profile information is generated again to determine the next data-set threshold. This process can be optimized by carrying out a threshold calculation during the application setup phase to reduce first-time invocation latencies. Threshold recalculation can also be motivated by the user, since we provide a mechanism for the user to manually stop the execution of a component if the response time becomes unreasonable. The Bifrost Performance Monitoring agent recalculates and re-sends profile information to the Bifrost Server. Simultaneously, the functionality is provided by the remote surrogate process.

This phase of component migration also takes into account the case of poor network quality. In this case, it might happen that: $t_{client} < t_{server} + t_{network}$, where t represents the time of component execution for a given data set. Thus, we need a reasonable estimate of how the network behaves, as explained in the next section.

3.3.2. Approximating Network Delay A number of tools can be used for approximating data transfer delay across the network (e.g., *ping* [29] and *tcpdump* [16]). In this study, we have followed an implementation similar to the trace-based routing methodology as suggested by Noble et. al. [31] to modify the *ping* utility to incorporate bottleneck delays. In the Bifrost implementation, we further smooth the delay values by incorporating a history of the observed behavior, and also take into account the overall quality of the network [32]. The delay then becomes: $d = \alpha d_n + (1-\alpha) d_{n-1}$ where d_n is the delay as measured at time t_n (or the current network snapshot), and d_{n-1} is the value of the previously measured delay. Different values of α were used to find a suitable range. We currently use $\alpha = 0.9$, thus placing more emphasis on recent information. A small lookup table for recently observed d_{n-1} values allows faster computation. This delay value allows us to compare client side vs. server side execution times. As of now we do not take into account possible pipelining of data across the network. Figure 4 summarizes the situation: When *BFCreateInstance()* is called or a COM call is re-directed, the registry is checked for an input data threshold value. If the input value is less than the threshold, then the local component is invoked. If no threshold exists, then either this is the first phase or the component was never migrated. If a local component is found, and there is no threshold, the performance agent is

invoked on the client. Finally, if there is no local component, then a remote pointer is obtained.

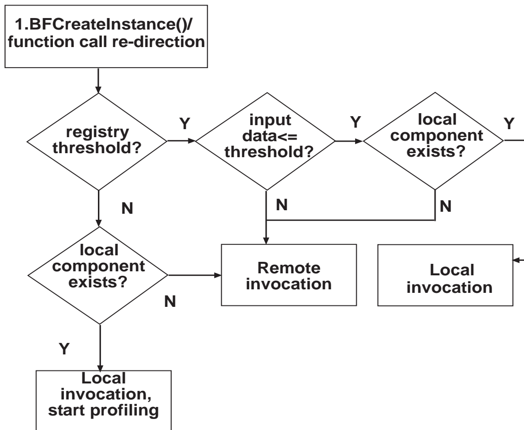


Figure 4. Migration decision flowchart

Although Bifrost is primarily designed to function in an "always connected" environment, disconnections may occur after a remote invocation resulting in significant delay. Bifrost addresses disconnection by leveraging upon component *affinities*. Component *affinity* is a scalar value that represents the extent to which a user or application invokes a particular component. In case of disconnection, components with high affinity that can run on the client device will already have been migrated to the local device. For components that have high affinity but can only run remotely, we propose to have lightweight components (if applicable) installed on the client device to be used only during disconnected state. An example of a lightweight component is an image processing tool that only supports gray scale image rendering as compared to a 24-bit color processing counterpart. If a lightweight component does not exist, then results of the remote computation might be lost during disconnection, resulting in re-computation upon subsequent reconnection. The Microsoft Transaction Service, as part of the COM+ services, provides a facility to set up event queues that decouple the client from the server and hold remote invocation events for each client. This allows results to be resent instead of recomputed in case of disconnection for most cases.

4. Results

In this section we present the performance benefits provided by the Bifrost component migration system, showing latencies for both local and remote invocations of a migrated component. The client devices used for these results were Sony Vaio (Pentium II, 233 MHz) laptops with 64 MB RAM utilizing an 802.11b-compliant wireless Ethernet NIC. The available disk space on the client machine was simulated as 100 MB for the purposes of component storage. The Bifrost server was a 700 MHz dual Pentium III Dell PowerEdge with 1 GB of RAM and a Fast Ethernet Interface.

4.1. Sample Application

We illustrate Bifrost component dispatching using a simple spell checker (developed locally) and an image processing (IMG) component [37]. These applications were chosen to illustrate particular features of Bifrost. For instance, a spell checker may be accompanied by a dictionary, which if of considerable size would play a significant role in the migration decision process. The IMG component on the other hand may not have a data set size problem, but it may consume significant processor resources on a smaller hand-held device. By evaluating the overhead of both server and client devices, Bifrost can dynamically determine the best execution site for these components.

4.2. System Response

This section presents a summary of the system response time when utilizing the facilities provided by the Bifrost library. We first present the overall performance benefit of being able to migrate a component rather than just having statically allocated remote components. This is followed by an evaluation of the impact of redirecting or wrapping component creation calls. Finally, we demonstrate a scenario that combines all these results to present the advantage of having an adaptive decision making system. We also take into account a loaded network, a loaded server, and a combination of both in our evaluation.

4.2.1. Overall System Response In this section, we present results for overall system response time. The sample application used here is the IMG component, which provides various image manipulation routines. For this example, the size of the input image file is 20 KB, and the time taken to execute the chosen image processing algorithm is 8.630 seconds on the server and 9.143 seconds on the client. In the case of a remote execution, we assume that the file is sent to the server and then the processed file is returned to the client (shown as network delay in this study). This is a worst-case situation, since with the help of a replication mechanism the amount of data to be reconciled between the client and server is significantly smaller. For this example, we consider various network loads, but assume that the server is not heavily loaded.

Figure 5 shows the overall benefit of using the Bifrost adaptive support when the image processing component runs on the local device as compared to running it remotely. The figure shows benefits of adaptation when the network

quality degrades due heavy network load. Note that "Bifrost Subsequent Calls" implies that the image processing window is still active, with the user invoking multiple image manipulation sessions.

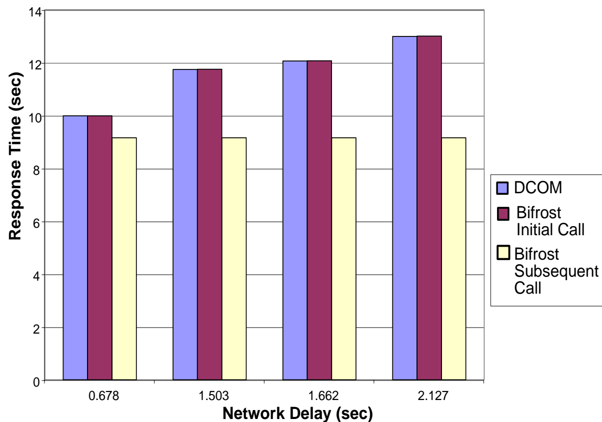
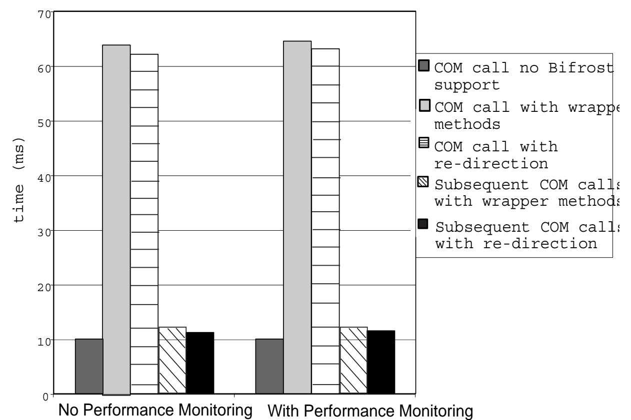


Figure 5. Overall system response

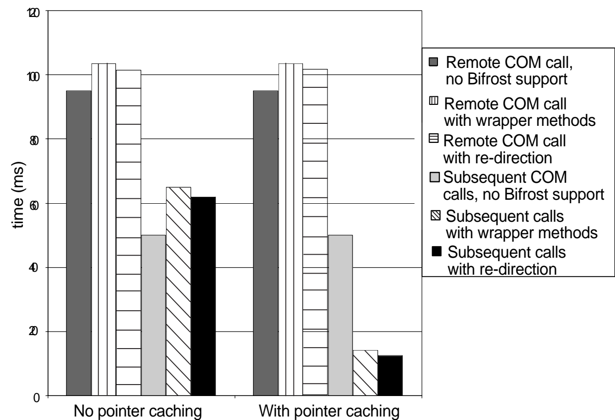
The first set of bars in Figure 5 represent the overall execution time when the network delay to transfer a 20 KB file is 0.678 sec. The network delay values used here have been empirically determined using the ideas presented in [3], in which the authors analyze file transfer latencies over a variety of client-server configurations. We have used values from four different configurations. The time taken to execute the component remotely with no Bifrost support is 10.1 seconds (labeled as "DCOM"). When Bifrost support is enabled, first-time initialization latencies cause a slight increase in the remote access time. However, after this invocation, the component has been migrated to the device (due to the Bifrost migration decision), and subsequent invocations are local. Overall local response time is 9.2 sec. Thus the overall improvement in running the component locally (due to adaptive support) when file transfer delay is 0.678 seconds is about 9%. In the final set of bars with file transfer delay of 2.127 sec, the improvement in response time over running the component locally (9.2 sec) vs. remotely (13 sec with no Bifrost support) is 29.2%.

4.2.2. Overhead For Local Invocation We now present an evaluation of the latency overhead due to the Bifrost library. The sample application used for this example is the spell checking component. To evaluate the effects of the wrapper APIs and function call re-direction, Figure 6(a) outlines the latency overhead during a local component invocation with Bifrost support (including system registry lookup) as compared to a normal invocation for the spell checker component. The first set of bars represent the measured time with no performance monitoring.

As indicated, a COM call to invoke a local component (DLL existing on the client machine) with no Bifrost support is around 10 ms on the client machine. There is of course no adaptive functionality enabled for such a call. When the Bifrost API methods are used (2nd and 4th set of bars above), there is an initial overhead of 64 ms (due to initialization of the Bifrost library and associated data structures) that decreases to 12 ms for subsequent calls to the same component. In the case of function re-direction (3rd and 5th set of bars), the initial latency is due to initializing the Detours library. Note that subsequent overhead for function re-direction is slightly less than the wrapper methods.



(a) Local Response



(b) Remote Response

Figure 6. System Response

The next set of bars in Figure 6(a) demonstrate the effect of performance monitoring. There is an increase in

first-time invocation latency to set up the performance counters (66 ms). Subsequent invocation times remain the same. With performance monitoring, the exit time of the application (e.g., the time taken for the component window to exit after the user presses “close”) is also increased (6 ms) since the statistic collection process needs to be stopped and results sent to the remote machine. This increase in exit time is not shown in Figure 6(a). This analysis does not take into account the execution time of the component (usually in seconds). Hence, the 10% - 20% (11 ms - 12 ms) overhead of the Bifrost library over normal COM (10 ms) call will likely become insignificant when execution time is factored in.

4.2.3. Overhead For Remote Invocation Figure 6(b) shows the latency overheads when the component being accessed is initially found on a remote server instead of the local machine. We cache the remote pointers locally to minimize the time taken to obtain the remote pointers and to reduce the network traffic on subsequent accesses to the interface pointers. In the internal implementation of DCOM, the runtime also caches remote pointers, but these are removed when the reference count of the COM object reaches zero. This results in higher latencies if the remote component is referenced later after the cached pointer has been deleted. We maintain these remote cached pointers until space limitations force their removal using an LRU replacement strategy. The fuzzy engine invocation time was observed to be 0.12 ms.

The first set of bars (“no pointer caching”) in Figure 6(b) represents the case in which the Bifrost system does not cache remote pointers. For a remote COM call or a DCOM call with no Bifrost support, the pointer retrieval time is dominated by the access time to the remote machine. When API wrapper methods or function re-direction is used via Bifrost, the initialization overhead is 6 ms - 8 ms. The last three bars for the “no pointer caching” case represent the **average** latency for subsequent remote accesses, and therefore take into account the case when the pointers have been removed from the DCOM runtime cache. While the fourth bar from the left (subsequent COM calls no Bifrost support) represents DCOM calls using pointers cached by the runtime, the last two bars are response times for subsequent calls with the Bifrost API leveraged upon the DCOM runtime caching.

The next set of bars indicate the improvements when we introduce remote pointer caching within the Bifrost system to augment the caching facilities used by the DCOM runtime. While the initialization latencies are the same, subsequent remote access latencies using the Bifrost library are reduced by about 77%. The response time for obtaining a remote pointer is reduced to about 14 ms using the Bifrost API, resulting in nearly identical access time to that of fetching a local pointer.

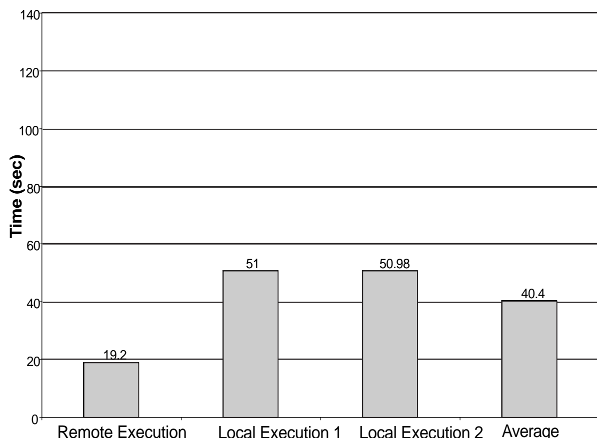
4.2.4. Determining the Best Execution Location Here we present an example that details the decision process in determining the best IMG component execution site. This example differs from the ones previously discussed in that the remotely-located component, although initially migrated to the local machine (since it fits on the device), may in fact have worse local response time. We use a different image processing algorithm that places more stress on the CPU. The migration decision takes into account the execution time of the IMG component both on the local machine and the remote machine. The input data file size in this case was again chosen to be 20 KB, and we have used network delay values collected by running the client within a wired network as well as a high latency wireless network. Initially we assume that the Bifrost server is located close to the client device on a wired LAN. For this example we also stress the server by running code to create artificial load on it. This is done to emulate the situation where many Bifrost users might be using the component services.

Figure 7(a) shows the decision-making process to determine the best execution location for the component based on response time seen by the user. The following points explain how the value for each bar in the figure is determined.

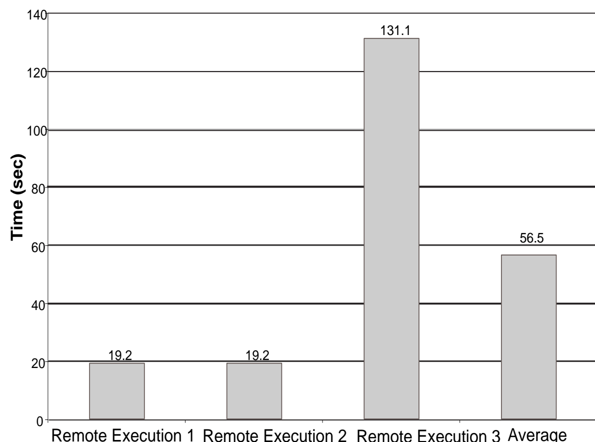
1. Remote Execution. In this case, the component is not present locally and remote execution is initiated with Bifrost support. Network delay to transfer 20 KB data is 0.625 sec, and the time taken to obtain the remote interface pointer is 102 ms. The execution time on the loaded server is 18.5 sec, giving an overall response time of 19.2 sec. The fuzzy decision engine is invoked on the server with the initial migration decision factor being *component size* $<$ *available space*. Hence, the component is migrated to the client device.

2. Local Execution 1. In this case, a subsequent access to the IMG component is local and the execution time for the component is 50.98 sec. The time taken to fetch local pointer is 17 ms (since profiling is initiated), giving an overall response time of 51 sec. The generated profiles are sent to the server for analysis and the fuzzy decision engine combines the new data determining the best execution location for the IMG component as remote. The threshold value for input data size as set by the server for the client is 10 KB. We have followed the common practice in the design of sliding window network protocols [39] whereby the windows size is halved when the network quality becomes poor. The size is then doubled on subsequent improvement.

3. Local Execution 2. Here we present the situation where the user moves to a wireless network, away from the Bifrost server. The network delay to transfer 20 KB of data was measured to be 112.5 sec. The network agent running on the client detects this change (using the net-



(a) Execution time with Bifrost



(b) Execution time without Bifrost

Figure 7. Best execution location

work delay model), and since a local component exists, the Bifrost Interface Manager chooses to use it. Profiling is not needed, and the response time is 50.98 sec. The threshold is set back to 20 KB.

4. Average response time. The average response time for these three accesses is computed as 40.4 sec.

Figure 7(b) shows the same sequence of accesses and the response time if no Bifrost support was present. To begin with, the execution is remote and the response time (without Bifrost library overhead) is 19.2 sec. Since the component is not migrated to the client, the next invocation (shown as "Remote Execution 2") also has a response time of 19.2 sec. However, when the user moves to a different location and the network delay increases, the response time degrades to 131.1 sec, with a 112.5 seconds delay for moving data. The average response time with no Bifrost support is thus 56.5 sec. Thus the improvement in the average response time is about 29% with the Bifrost adaptive support in this scenario. Note that this improvement is calculated when the user invokes the component locally (during high network latency) only once. Subsequent local invocations under these conditions would further improve the average response time as compared to running the component remotely.

5. Related Work

The Bifrost Interface Management API is the first system to build upon an existing distributed systems technology with the intent of incorporating an adaptive decision capability to evaluate the best execution site of application components. Adaptive component migration has been explored in the design of Spectra [11], and has principles similar to the work done for Bifrost (adaptive functionality, decision engine, etc.). However Bifrost differs in the metrics that are used to evaluate the best execution site, and in the generality of the approach taken by Bifrost. As compared to Spectra, we provide options of a more general purpose API, or the ability to instrument methods using Detours [15], which can be integrated with any application designed for a platform supporting DCOM. Spectra builds upon existing support provided by Odyssey [32], which uses the notion of "data fidelity" to tune applications while using a gradient descent decision-making mechanism.

Remote execution has been widely researched at lower levels of granularity such as thread migration [35], however, here we comment only on work closely related to the work done for Bifrost. Fuzzy logic techniques to improve performance of operating system services have been discussed in [18]. In this paper, the authors apply fuzzy logic techniques to extend the file system semantics.

The problem of designing an adaptive system such as that found in Bifrost can be approached in several different ways [5]. Distributed systems technologies such as CORBA [33] and DCOM [22], which serves as the underlying system on which we have built Bifrost, only provide transparent object invocation across different platforms. The Coign ADP system [14], built on COM, statically evaluates good component distribution. Toolkits such as Rover [17] have been developed to facilitate adaptive behavior. Although Bifrost does not provide such a toolkit, the Bifrost libraries can be used with a system such as Coign by making runtime decisions on the execution location of partitioned components. Some adaptive systems use Quality of Service (QoS) as the underlying metric without leveraging on explicit OS support [4, 21, 30]. Although we do not research means to improve QoS guarantees, the decision making mechanism is aware of the

quality of the network in which the client device operates.

Adaptability can also be implemented by providing programming language changes to emphasize the development of middleware constructs to support code and data mobility [1, 12, 38, 42]. There are no new programming language constructs developed for Bifrost. We either wrap component creation calls or use runtime instrumentation to return the appropriate interface pointer to the application. Finally, some adaptive applications classified as "Reflective Middleware" include the ability to inspect and analyze their behavior based upon a causally-connected self representation (CCSR) [9]. The HADAS [36] system is one such example, providing an infrastructure for object interconnectivity, security, and persistence. Currently Bifrost does not use reflection as a means to adapt the application due to lack of explicit support in the currently used code-base. The .NET environment [23] recently available from Microsoft can allow some reflective capability to exist. We will investigate these capabilities as the area matures.

6. Conclusions and Future Work

The Bifrost adaptive component management system seeks to reduce the latency of component invocations for distributed applications. While using a distributed systems technology (DCOM) for underlying support, Bifrost also utilizes operating system services for creating a profile to guide component placement for improved performance.

We have presented the Bifrost framework for adaptively migrating components based on multiple performance parameters. By taking into account aspects of the client and server architecture (e.g., the processor speeds, available RAM, and disk space), the network latency separating the client and the associated Bifrost server, and the data size involved in the component's functionality, the Bifrost Component Migration facility is able to efficiently make decisions regarding the best placement for a given component. The Bifrost decision-making mechanisms applied on application templates have shown real performance results showing reduction in response time for the user of up to 29%.

With the basic adaptive framework having been established, we are now looking at further extending the system to commercial applications with exposed COM or DCOM interfaces (e.g., Microsoft Word or PowerPoint), as well as porting our system to the Windows CE environment.

We are also working on integrating affinity values of application components to devices, and are developing various affinity representation models based on data access patterns through custom file tracing and URL logging tools. We use affinity to decide when and where to automatically move data (files). As a user moves, affinity is used to move the user's data to reduce access latency. Subsequent accesses, and the locations from which these accesses take place, change affinity relationships between data, users, and locations. Affinity values are built up when files are accessed, while affinities are aged until the next access. Similar work with the intent of bringing recently used data close to the user has been implemented in systems such as SEER [19] to *hoard* data. A file may only be accessed while the user is at one location and therefore only would be hoarded when the user is currently at that location. Unlike our affinity-based approach, such systems do not handle data that is location-dependent. Affinities can also be used to represent component access patterns and component usage history, which enables us to introduce a component pre-fetch mechanism to further improve the response time for the user.

Acknowledgments

This work is supported in part by a grant from Microsoft Research, NSF Award ANI-0125987, and the Intelligent Information Systems Institute at Cornell University.

References

[1] A. Acharya, M. Ranganathan, and J.Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. *Mobile Object Systems, Lecture Notes in Computer Science*, pages 111-130, No. 1222, April 1997.

[2] M. Avvenuti and A. Veccio. Embedding Remote Object Mobility in Java RMI. *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, October 2001.

[3] P. Barford and M. Crovella. Measuring Web Performance in the Wide-Area. *IMA "HOT TOPICS" Workshop: Scaling Phenomena in Communications Networks*, October 1999.

[4] F. Baschieri, P. Bellavista, and A. Corradi. Mobile Agents for QoS Tailoring, Control and Adaptation Over the Internet: The UbiQoS Video on Demand Service. *Proceedings of the 2002 Symposium on Applications and the Internet*, January 2002.

[5] K. Bergner, R. Grosu, et al. Focusing on Mobility. *Proceedings of the 32nd Hawaii International Conference on System Sciences*, January 1999.

[6] A. Bhatnagar. Adaptive Component Management in the Bifrost System (M.S. Thesis). *Cornell Computer Systems Lab Technical Report CSL-TR-2002-1026*, August 2002.

[7] P. Bhagwat. Bluetooth: Technology for Short-Range Wireless Apps. *IEEE Internet Computing*, Vol. 5, No. 3, May-June 2001.

[8] B. Cosco. *Neural Networks and Fuzzy Systems.* Prentice Hall, 1992.

[9] G. Coulson. "What is Reflective Middleware?" `http://dsonline.computer.org/middleware/RMarticle1.htm`.

[10] C. Elkan. The Paradoxical Success of Fuzzy Logic. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 698-703, August 1993.

[11] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[12] R.S. Gray. Agent TCL: *Alpha Release 1.1, 1995*.

[13] J. Gomez, A. Campbell, M. Naghshineh, and C. Bisdikian. PARO: Conserving Transmission Power in Wireless Ad-hoc Networks. *Proceedings of the IEEE 9th International Conference on Network Protocols (ICNP'01)*, November 2001.

[14] G. Hunt. Automatic Distributed Partitioning of Component-based Applications. Ph.D. thesis, University of Rochester, August 1998.

[15] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[16] V. Jackobson, C. Leres, and S. McCanne. The Tcpdump Manual Page. Lawrence Berkley Laboratory, Berkely CA.

[17] A. Joseph and F. Kaashoek. "Building Reliable Mobile-aware Applications Using the Rover Toolkit." `http://www.pdos.lcs.mit.edu/rover/`.

[18] A. Kandel, Y.-Q. Zhan, and M. Henne. On the Use of Fuzzy Logic Technology in Operating Systems. *Journal on Fuzzy Sets and Systems*, pp. 241-251, Vol. 99, No. 3, 1998.

[19] G. Kuenning and G. Popek. Automated Hoarding for Mobile Computers. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[20] E. de Lara, D. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[21] S. Lu and V. Bharghavan. Adaptive Resource Management Algorithms for Indoor Mobile Computing Environments. *Proceedings of the SIGCOMM*, August 1996.

[22] Microsoft DCOM Website. `http://www.microsoft.com/com/tech/dcom.asp`.

[23] Microsoft .NET Website. `http://msdn.microsoft.com/netframework/`.

[24] COM/DCOM Security Implementation. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceipc/htm/dcom~8.asp`.

[25] Microsoft Windows Kerberos Implementation. `http://www.microsoft.com/windows2000/techinfo/howitworks/security/kerberos.asp`.

[26] Protected Extensible Authentication Protocol. `http://www.microsoft.com/windowsxp/pro/techinfo/administration/wirelesssecurity/improvedsolutions.asp`.

[27] MSDN Product Support Website. `http://www.msdn.microsoft.com` Search: Profiling Windows NT services (Q117681).

[28] Microsoft IntelliMirror Technology Documentation. `http://www.microsoft.com/windows2000/techinfo/howitworks/management/intellimirror.asp`.

[29] M. Muus. Ping Documentation. `http://ftp.arl.mil/~mike/ping.html`.

[30] T.Nakajimaa and H. Aizub. Middleware for Building Adaptive Migratory Continuous Media Applications. *International Journal of Software Engineering and Knowledge Engineering*, pages 83-107, Volume 11, Issue 1, February 2001.

[31] B. Noble, M. Satyanarayanan, G. Nguyen, and R. Katz. Trace-based Mobile Network Emulation. *Proceedings of ACM SIGCOMM*, September 1997.

[32] B. Noble, M. Satyanarayanan, et al. Agile Application-aware Adaptation for Mobility. *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[33] Object management group website. `http://www.omg.org`.

[34] J. Richter. *Programming Applications for Microsoft Windows (Fourth Edition)*, Microsoft Press, Redmond Washington 1999.

[35] H. Abdel-Shafi, E. Speight, and J. Bennett. Efficient User Level Thread Migration and Checkpointing on Windows NT Clusters. *Proceedings of the* 3rd *Usenix Windows NT Symposium*, July 1999.

[36] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic Adaptation and Deployment of Distributed Components in Hadas. *IEEE Transactions on Software Engineering*, pages 769-787, Vol 27, No 9, September 2001.

[37] IMG Component Website. `http://www.smalleranimals.com/development.htm`.

[38] N. Suri, J. Bradshaw, et al. An Overview of the NOMADS Mobile Agent System. *In Proceedings of the European Conference on Object-oriented Programming*, June 2000.

[39] A. Tanenbaum. *Computer Networks (Third Edition)*. Prentice Hall, 1996.

[40] T. Thai. *Learning DCOM*. O'Reilly and Associates, 1999.

[41] C. Toh. Maximum Battery Life Routing to Support Ubiquitous Mobile Computing in Wireless Ad Hoc Networks. *IEEE Communications*, June 2001.

[42] B. Venners. "Under the hood: Architecture of Aglets". `http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html`. JavaWorld, April 1997.