Transparent Network Connectivity in Dynamic Cluster Environments

Xiaodong Fu, Hua Wang, and Vijay Karamcheti
Department of Computer Science
New York University
{*xiaodong, wanghua, vijayk*}@*cs.nyu.edu*

Abstract

Improvements in microprocessor and networking performance have made networks of workstations a very attractive platform for high-end parallel and distributed computing. However, the effective deployment of such environments requires addressing two problems not associated with dedicated parallel machines: heterogeneous resource capabilities and dynamic availability. Achieving good performance requires that application components be able to migrate between cluster resources and efficiently adapt to the underlying resource capabilities. An important component of the required support is maintaining network connectivity, which directly impacts on the transparency of migration to the application and its performance after migration. Unfortunately, existing approaches rely on either extensive operating system modifications or new APIs to maintain network connectivity, both of which limits their wider applicability.

This paper presents the design, implementation, and performance of a transparent network connectivity layer for dynamic cluster environments. Our design uses the techniques of API interception and virtualization to construct a transparent layer in user space; use of the layer requires no modification either to the application or the underlying operating system and messaging layers. Our layer enables the migration of application components without breaking network connections, and additionally permits adaptation to the characteristics of the underlying networking substrate. Experiments with supporting a persistent socket interface in two environments—an Ethernet LAN on top of TCP/IP, and a Myrinet LAN on top of Fast Messages—show that our approach incurs minimal overheads and can effectively select the best substrate for implementing application communication requirements.

keywords Workstation cluster, process migration, network connectivity, adaptation, API interception

1 Introduction

With improvements in microprocessor and networking performance, cluster environments have become an increasingly cost-effective option for general parallel and distributed computing. Despite demonstrations of effectiveness in controlled situations (e.g., a dedicated cluster of workstations employed for scientific computations), wider-scale use of cluster environments for general applications has been hampered by the need to handle two characteristics of such environments: *heterogeneous resource capabilities* and *dynamic availability*. The former is a consequence of both the incremental construction of cluster installations over an extended time period and the emerging trend towards integrating mobile resource-constrained devices in such environments. The latter characteristic primarily arises due to the fact that cluster environments are not dedicated to a single application to permit optimal use of shared resources.

The traditional method of addressing these characteristics relies on being able to migrate application components among the cluster resources and efficiently adapting to the underlying resource capabilities. Such migration capability effectively supports the distribution of computational load among nodes in a cluster, the dynamic addition and removal of computing nodes to a running application, and the migratability of applications between fixed and mobile devices. However, to fully realize these benefits, the migration must be supported as transparent to the application as possible. In particular, network connectivity must be maintained, both within application components and between the application and the outside world. Unfortunately, maintaining connectivity is complicated by the fact that (1) sending and receiving network messages changes the state of operating system (OS) buffers, and (2) an application typically internalizes several operating system handles (such as socket identifiers and IP addresses), which stop being relevant upon migration.

Existing approaches deal with the above problems either by relying on extensive modifications to OS structures to support migration [6, 12, 13, 10], or by requiring the use of a new application programming interface (API) [2, 4, 3, 7] whose implementation isolates the application from the consequences of migration. Neither of these choices is ideal because they cannot be applied to existing OSes and applications. Moreover, most such solutions do not address the issue of adapting to changes in resource characteristics (e.g., the availability of networking substrates with different capabilities).

In this paper we present the design, implementation, and performance of a network connectivity layer that addresses the above shortcomings. Our layer operates *transparently* between the application and the underlying communication layer or operating system (our specific context is Win32 applications running on top of Windows NT). This layer interfaces with the application and the operating system using API interception techniques [1, 8], which permits calls made to system APIs to be diverted to a set of routines provided by our layer. This facility permits our layer to maintain network connectivity across application component migrations without requiring modification to either the applications that sit on top, or the communication layers and operating systems that sit below. In particular, this network connectivity layer manages the mapping between physical and virtual handles and the uninterrupted transfer of required state whenever a component is migrated.

Moreover, using the same techniques, this layer can also support the dynamic adaptation of the application to changing underlying resource characteristics. For instance, upon detecting that the network connection after migration is slow, the layer can transparently introduce compression and decompression steps at the two ends of the connection, thereby trading off additional processing for network bandwidth. Thus, this layer provides a natural place for incorporating several policies for customizing application use of underlying resources.

To assess the complexity and performance overheads of the network connectivity layer, we describe its implementation in the concrete context of two environments—an Ethernet LAN on top of TCP/IP, and a Myrinet LAN on top of Fast Messages [11, 9]. Our results show that the layer incurs minimal overheads and can effectively select the best substrate for implementing application communication requirements.

The rest of this extended abstract is organized as follows. Section 2 presents relevant background and related approaches for maintaining network connectivity. Section 3 presents the design and implementation details of our transparent communication layer. The performance overheads of the layer are analyzed in Section 4. Section 5 discusses the applicability of our approach to other related problems and we conclude in Section 6.

2 Background

The context for this research is applications written to the Win32 and WinSock interfaces running on top of the Windows NT operating system. Our goal is to provide transparent network connectivity across migration of such applications, and constitutes one component of a larger research project called *Computing Communities (CC)* [5]. CC articulates a novel method of middleware development, which does not suffer from the excruciating problem of having to redesign, recode, or even recompile the applications. The binaries of all existing applications can run on a new, distributed platform without modification. This distributed platform realizes a "computing community", in which all of the physical resources such as CPU, display, file system, network are virtualized and provide the application with a view of running on a virtual multiprocessor system.

2.1 Related Work

Previous approaches for maintaining network connectivity fall into two broad categories: modifications to the OS network protocol stack, and introduction of a new API for accessing underlying resources.

Modifying the OS network layer. Several researchers [13, 12, 6] have successfully demonstrated transparent network connectivity by across process migrations by incorporating changes to kernel data structures and protocols. For instance [13, 12], the implementation of network migration on top of the Chorus operating system modifies the network manager so that a migrating process' ports are marked as migrating. Messages sent to the port of a migrating process results in the requesting node being informed of the migration, causing the request to be reissued. While such solutions provide required functionality, their reliance on kernel modifications restricts their applicability in the case of commodity OSes.

Modifying the API interface. An alternative approach isolates the application from changes in its mapping to underlying resources. Typically, this requires modifying application abstractions using new APIs. For instance, the application can use only connectionless protocols (using global target identifiers that are guaranteed to remain unchanged across migrations) or fault-tolerant group communication primitives [3, 7]. Some other systems rely on appropriate run-time support to construct a global name space for all structures [2, 4] where the application remains unaware that its mapping to underlying physical resources has changed. The primary handicap of such approaches is their limited applicability to commodity applications that are written using standard APIs.

2.2 API Interception

Our approach addresses the above shortcomings by maintaining network connectivity using a user-level middleware layer that is *transparently* inserted between the application and the underlying OS; neither the application nor the OS needs to be modified. In fact, the application does not even need to be recompiled or relinked.

Our middleware layer is inserted using a recently developed technique [1, 8] called API interception. This technique relies on a run-time rewrite of portions of the memory image of the application (either the import table for functions in dynamically-linked libraries (DLLs) or the headers of arbitrary functions) to redirect API requests originating from the application to appropriate functions in the middleware layer. This paper describes how this basic mechanism augmented with support for handle translation, buffering, and flow control can be used to provide network connectivity across migration of application components. An additional advantage of our approach is that it can *adapt* transparently to changes in the underlying network configuration. For example, given a situation where the application components have a choice of multiple networking substrates to communicate over (e.g., both Ethernet and Myrinet), the middleware layer can automatically redirect application interactions to the network (and accompanying underlying messaging layer) that delivers the higher performance. Note that this switch between networks and messaging layers is accomplished completely transparent to the application.

3 Persistent Network Connections: Design

To make a network connection transparently migratable and capable of adapting itself to underlying resource characteristics, the communication layer *virtualizes* the physical socket connection. Virtualization, achieved using API interception, comprises of two parts: (1) *association of a global identity* (GID) with the connection independent of the physical location of the end-point processes, and (2) *rerouting of application requests* that use the socket to appropriate handler routines, which complete the requests using available physical resources. Figure 1 shows these two virtualization components.

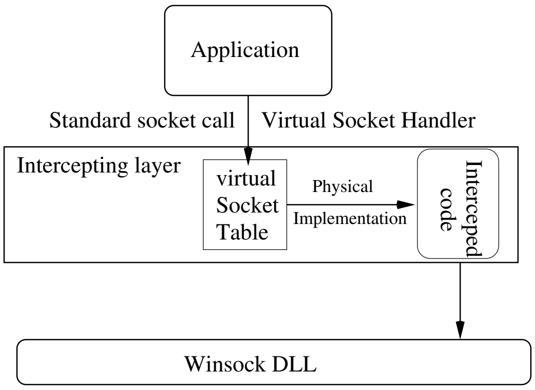


Figure 1: Virtualizing the socket layer using API interception.

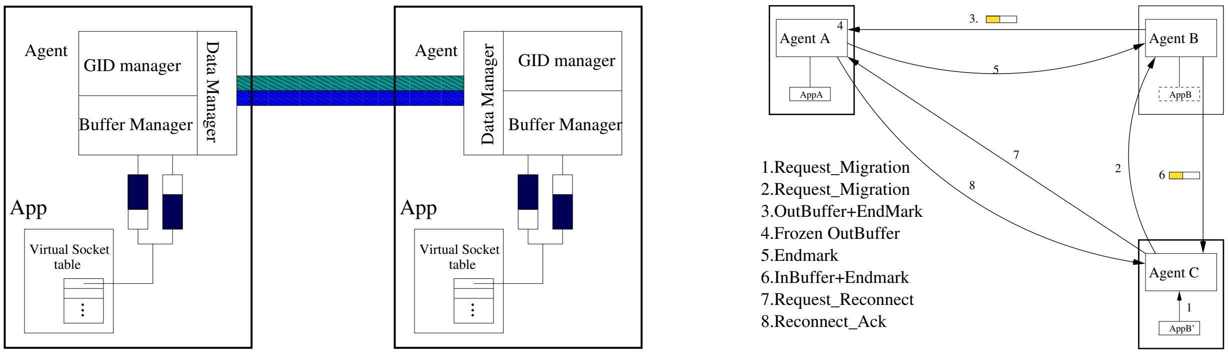
The GID-to-physical socket translation and the handlers of redirected requests together constitute the *agent* activity of the layer. The agent creates virtual sockets by allocating appropriate physical resources and associates a GID with it. Application requests use this GID and are handled by realizing them using semantics-preserving operations on the underlying physical network resources. The GID persists across migrations of application components: the agent allocates physical resources on the new location and reassociates the GID with them. Agents on the two nodes involved in the migration coordinate with each other to ensure that application components remain unaware of the migration. In addition to performing GID translations and physical data transfer, the agent handles flow control and management of resources at the two ends of the connection.

This section describes two implementations of this design. The two implementations, referred to as the *thick agent* and *thin agent* implementations in the rest of the paper, differ in how the agent activity is integrated with the application and represent different tradeoffs between extensibility and performance. In the thick agent implementation, described in Section 3.1, the agent is realized as a separate process that interacts with the application components using a pair of FIFO buffers. In the thin agent implementation, described in Section 3.2, a subset of the agent functionality that is on the critical path is injected into the application itself (again using API interception). The rest of the agent functionality remains in a separate process that is only used to coordinate actions at startup and upon migration and interacts with the application components using a shared buffer.

In both cases, we assume that the actual migration of the application components themselves is accomplished through orthogonal mechanisms not discussed here. These mechanisms can be simple such as restarting an application component on a new node with different parameters (a common strategy for stateless servers), or more elaborate involving process checkpoint and restart.

3.1 Thick Agent Implementation

The thick agent implementation is illustrated in Figure 2. The basic idea is that each end of a socket connection (in general, any interprocess communication (IPC) mechanism) can be abstracted in terms of a pair of FIFO buffers between the application component and the agent process. The IPC mechanisms inject data into and extract data from these buffers; physical data transfer is realized by the agent processes. The agent processes are also responsible for buffer management and maintaining data stream continuity upon migration.



(a) Overall Structure

(b) Migration

Figure 2: Overall structure of thick agent and migration protocol

Buffer Management The FIFO buffers between the application component and the agent processes are of fixed size. The agent processes remove messages from the application out-buffer and inject them into the network, and extract messages from the network and insert them into the application in-buffer. To handle situations where the application is not responsive in consuming messages in the in-buffer, the agent processes divert received messages into dynamically allocated overflow buffers prior to transferring them to the in-buffer.

Data Stream Continuity To maintain data stream continuity across migrations, both the data stored in the FIFO buffers on the original site as well as any messages in transit must be flushed to the new site. The agent processes coordinate to achieve this using the following eight steps (Figure 2(b) shows these steps for the migration of a connection end-point from node B to node C; the other end-point stays fixed on node A):

1. Upon realizing that it has migrated, the application component sends a REQUEST-MIGRATION message with the connection GID as a parameter to the local agent (on node C).
2. The agent on node C forwards the message to the agent at the old site (node B).
3. Upon receiving the REQUEST-MIGRATION message, the agent on node B flushes its out-buffer to the other end of the connection (node A) and sends an END-MARK message. It then waits for Step 5, extracting in-transit messages into its in-buffer.
4. When the agent on node A receives the END-MARK message, it freezes the out-buffer for the connection.
5. The agent on node A then injects an END-MARK message into the network.
6. Receipt of this END-MARK message on node B implies that no more data will be sent from node A on this connection. The agent then forwards all of the messages in its in-buffer to node C.[1]
7. When the agent on node C receives all of these messages, it recreates its data structures and sends out a REQUEST-RECONNECT message to the agent on node A.
8. Upon receiving the REQUEST-RECONNECT message, the agent on node A reactivates the send buffer for the connection, and sends back an ack message to node C.

The primary advantage of the thick agent implementation is its extensibility. Support for new IPC mechanisms can be easily incorporated with minimal code modifications to capture its semantics. Another advantage

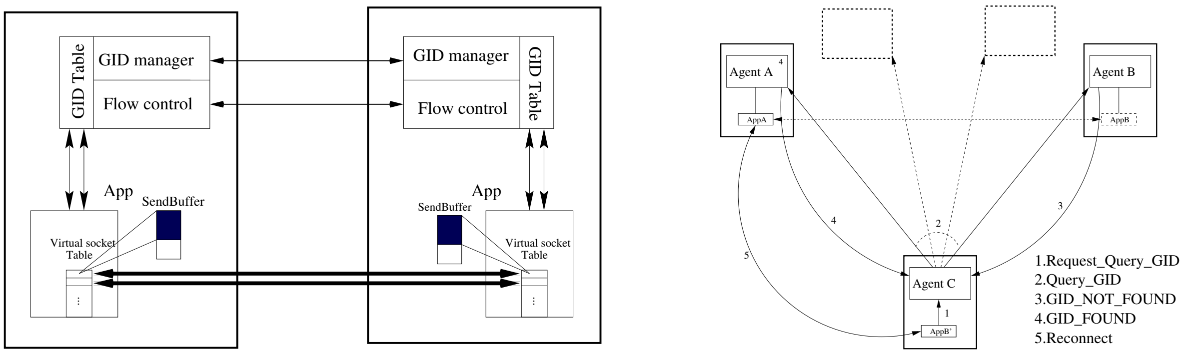
[1] The message forwarding actually proceeds concurrently with steps 4 and 5, but is stated this way for clarity.

is the complete decoupling between application-agent and agent-agent interactions. This decoupling permits the agents to appropriately adapt to the underlying communication substrate without affecting the rest of the implementation in any way. For instance, the agents can use a faster networking layer/substrate (e.g., FastMessages on Myrinet) when available, or introduce codecs to minimize bandwidth requirements on a wireless connection.

The primary disadvantage of this implementation is its performance penalty. Each data transmission between data components will introduce two extra context switches between the application and agent processes and two extra data copies. To improve on this, in the thin agent implementation described next, we move the agent data transfer functionality into the application itself. This reduces the overhead to a single data copy on the send side.

3.2 Thin Agent Implementation

The basic idea of the thin agent implementation, illustrated in Figure 3, is to move the critical part of the agent functionality into the application component itself. Note that the application binary remains unmodified; this functionality is injected at load-time using the API interception technique described earlier. As a consequence, applications communicate with each other directly, and are responsible for maintaining the state of the data connection and for adapting themselves to underlying resources. Application components detect migration by detecting that the existing connection has broken. To re-establish a connection, they rely on GID tables of active local connections maintained in the separate agent process at each site. Agents coordinate upon migration to determine the two connection end-points; application components reconnect to these points by replaying the original (logged) API request.



(a) Overall Structure

(b) Migration

Figure 3: Overall structure of thin agent and migration protocol

Buffer Management Each end of the connection, besides storing necessary state information, also maintains a send-side buffer. This buffer stores copies of messages that have been transmitted but not yet acknowledged. Send-side buffering suffices because data is directly routed into application buffers on the receive side. Management of these buffers uses a straightforward window-based flow control scheme. These buffers are expanded dynamically as required by application communication patterns; space is freed upon acknowledgement of the receipt of the corresponding message(s) on the receiver. For efficiency, acknowledgements are batched together and piggy-backed on data messages. For improved locality and better small message performance, messages below a threshold size are copied inline into the send buffer; all other messages are allocated out-of-line (with explicit freeing of storage upon acknowledgment of receipt).

Data Stream Continuity To maintain data stream continuity, the application components need to reconnect after migration. The send buffers associated with the virtual socket connection at each end point contain sufficient state to handle any retransmissions as necessary. Note that migration of an application component is assumed to also migrate the corresponding send buffer(s).

Reconnection of application components is achieved using the following five-step migration procedure (Figure 3(b) shows these steps for the migration of a connection end-point from node B to node C):

1. The migrated application component, upon detecting a broken connection, sends a REQUEST-QUERY message with its GID as a parameter to the local agent (on node C). The other end-point of the connection (on node A) also performs a similar action (with its local agent).
2. The agent on node C coordinates with other agents (e.g., using multicast) to determine the other end-point of the connection.
3. The agent on the old site (from where the application component) will no longer find the process corresponding to the GID, hence will not respond.
4. The agents on nodes A and C will receive information about the location of each other's end-points (node C and A respectively).
5. The application components on nodes A and C (a) reconnect with each other, (b) exchange state information (about messages sent and acknowledged), (c) retransmit any lost messages using the send buffers, and (d) resume operation.

The primary advantage of the thin agent implementation is its efficiency; the only overheads that remain are for the send-side buffering. However, this advantage comes at the cost of increased complexity: the agent functionality injected into the application must be aware of the underlying resource characteristics and explicitly adapt to them. As mentioned earlier, this complexity does not affect the user application.

3.3 Using Other Transport Layers

Although we have discussed the thick and thin agent implementations on top of reliable connection-based transport protocols, they can be as conveniently implemented on top of other transports. We briefly sketch the differences for two interesting transports. For *unreliable connectionless layers* (e.g., UDP), the agent activity must also handle retransmission and removal of duplicates. In addition, the agents need to coordinate to locate the end-points of the connection, similar to what was described above for the thin agent implementation. For *active message layers* (e.g., FM), in addition to the above, the agent activity needs to handle the extraction of messages that belong to other application streams. An efficient implementation minimizes the amount of buffering that needs to be provided, directly rerouting received messages into posted buffers when the latter are available.

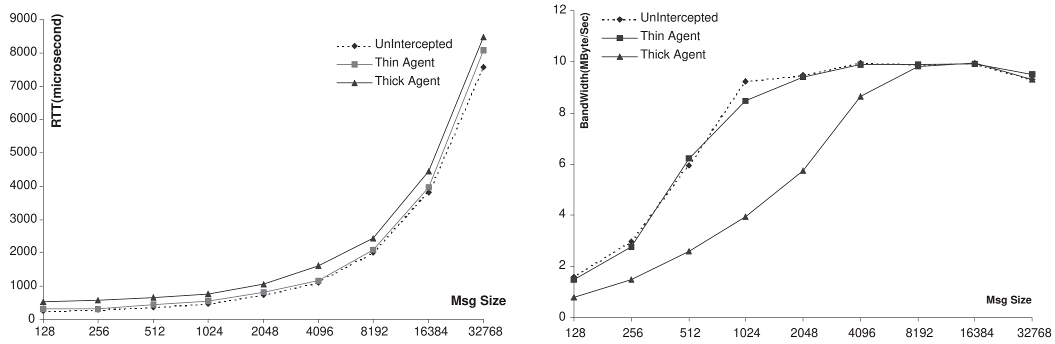
4 Persistent Network Connections: Performance

To assess the run-time overheads of the network connectivity layer, we measured the performance of the thick and thin agent implementations in two environments—a 100 Mbps Ethernet LAN on top of TCP/IP, and a 1.28 Gbps Myrinet LAN on top of Illinois Fast Messages (FM). All of the experiments were run on Pentium Pro 200 MHz machines with 64 MB memory. The FM experiments used the HPVM 1.2 release of the messaging layer.

NOTE TO REVIEWERS: This extended abstract only reports on latency and bandwidth microbenchmarks. The full paper will contain additional performance results for a stateless server application that takes advantage of our connectivity layer.

4.1 Overheads of Transparent Connectivity

Figure 4 shows the impact on round-trip time and bandwidth of maintaining a transparent migratable connection. Tables 1 show the raw data for these plots.



(a) Round-trip times vs. message size

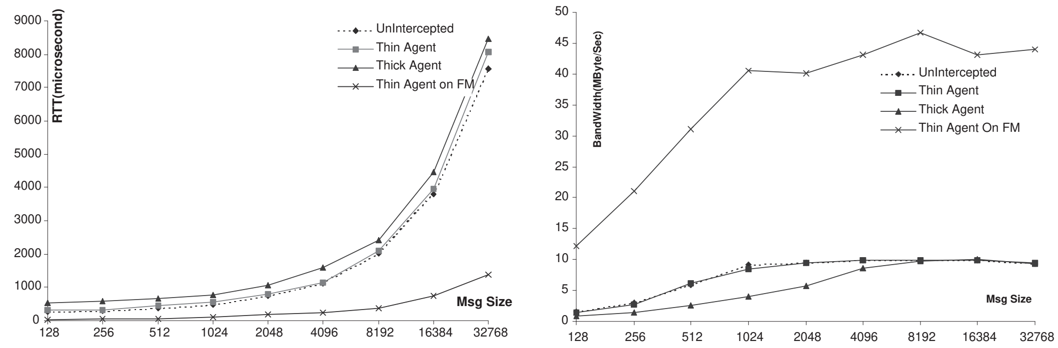
(b) Bandwidth vs. message size

Figure 4: Round-trip times and bandwidth achieved by the thick and thin agent implementations in the Ethernet/TCP environment.

Message bytes	128	256	512	1024	2048	4096	8192	16384	32768
Round Trip Time(microsecond)									
Unintercepted	265	294	362	470	753	1104	2013	3804	7569
Thin agent	316	318	449	563	808	1152	2094	3958	8081
Thick agent	538	577	655	773	1066	1603	2424	4458	8459
BandWidth(MBytes/sec)									
Unintercepted	1.58	2.97	5.94	9.24	9.46	9.95	9.89	9.92	9.30
Thin agent	1.48	2.77	6.23	8.46	9.42	9.89	9.89	9.94	9.52
Thick agent	0.81	1.49	2.59	3.95	5.74	8.63	9.81	9.96	9.30

Table 1: Round-trip time and Bandwidth vs. message size for the thick and thin agent implementations in the Ethernet/TCP environment.

The plots show that the thick agent implementation has measurable impact on both round-trip time and bandwidth (increasing the former by up to 24% and decreasing the latter by up to 50%), primarily because of additional context switches and data copies. On the other hand, the thin agent implementation incurs no noticeable overheads as compared to the unintercepted (and hence not-migratable) TCP/IP implementation. Both round-trip time and bandwidth are within 5% of the unintercepted version, demonstrating that our connectivity layer can efficiently maintain data stream continuity over migrations.



(a) Round-trip time vs. message size

(b) Bandwidth vs. message size

Figure 5: Round-trip times and bandwidth achieved by the thin agent implementation in the Myrinet/FM environment.

Message bytes	128	256	512	1024	2048	4096	8192	16384	32768
Round Trip Time(microsecond)									
Unintercepted	265	294	362	470	753	1104	2013	3804	7569
FM-thin-agent	37	47	65	100	181	250	373	738	1367
Bandwidth(MBytes/sec)									
Unintercepted	1.58	2.97	5.94	9.24	9.46	9.95	9.89	9.92	9.30
FM-thin-agent	12.20	21.02	31.06	40.60	40.07	43.14	46.67	43.07	44.05

Table 2: Round-Trip time and Bandwidth vs. message size for the thin agent implementation in the Myrinet/FM compared with Unintercepted socket interface in the Ethernet/TCP environment.

4.2 Adaptation to Networking Substrate

As mentioned earlier, our communication layer can transparently adapt to the characteristics of the underlying networking susbtrate. To assess the gains possible by such adaptation, we compared the round-trip time and bandwidth of the thin agent implementation in the Myrinet/FM environment with the corresponding implementation in the Ethernet/TCP environment. Figure 5 and Tables 2 show these costs.

The plots show that the mismatch between application-level WinSock semantics and transport-level FM semantics results in higher overheads in the communication layer as compared to the Ethernet/LAN environment. To put our implementation in context, base FM has a minimum round-trip time and maximum bandwidth of 21μs and 65 MB/s on our experiment testbed. In contrast, our layer achieves a minimum latency of 35μs and a maximum bandwidth of 46 MB/s. While these numbers by themselves are quite good, we expect them to improve further with additional tuning of our implementation.

More importantly, the plots show the advantages of our layer automatically adapting itself to the underlying substrate. The thin agent implementation on Myrinet/FM improves round-trip times by up to 8x, and bandwidth by up to 5x as compared to the Ethernet/TCP environment. These improvements become transparently available to the application components.

5 Discussion

Although we have limited our attention here to providing data stream continuity across migrations, our approach of transparently rerouting application requests to a middleware layer can also be used to address several related concerns. We briefly discuss some of these issues below.

Separation from the underlying OS interfaces. The layer can be used to decouple an application from the interfaces provided by the underlying operating system. The implementation of our communication layer on top of the FM interface demonstrates this capability; application-level WinSock requests are translated to semantically equivalent sets of FM operations.

Adaptation to changing resource characteristics. The middleware layer provides a natural place for incorporating different policies for customizing application use of underlying resources. This provides a powerful infrastructure for allowing the application to become aware of changes in network conditions and adapt to them. These adaptation policies can be either application-independent (e.g., interfacing with different transport layers or inserting compression/decompression operations at the end-points), or application-aware (e.g., selective dropping of packets in a video stream based on its encoding to reduce overall bandwidth requirements).

6 Conclusion

We have described the design of a communication layer that maintains network connectivity across migrations of application components in a distributed system. This layer is transparently inserted between unmodified applications and commodity operating systems using API interception techniques. Results based on implementations of the layer in two environments—Ethernet on top of TCP/IP and Myrinet on top of FM—show that the layer introduces negligible overheads during normal operation (when the components do not migrate), and can additionally seamlessly choose the best among available networking substrates.

References

[1] Balzer, R. Mediating Connectors. In *Proc. of ICDCS Middleware Workshop*, 1999.

[2] Baratloo, A., Dasgupta, A., and Kedem, Z. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proc. of 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.

[3] Birman, K. Replication and fault-tolerance in the ISIS system. In *Proc. of 10th ACM Symp. on Operating System Principle*, pages 79–86, 1985.

[4] Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., and Zhou, Y. Cilk: An efficient multi-threaded runtime system. In *5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[5] Dasgupta, P., Karamcheti, V., and Kedem, Z. Transparent distribution middleware for general purpose computations. In *Proc. of Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.

[6] Douglis, F. and Ousterhout, J. Process migration in the sprite operating system. In *Proc. of 7th Intl. Conf. on Distributed Computing Systems*, pages 18–25, 1987.

[7] Hayden, M. The Ensemble System. Technical Report TR98-1662, Cornell University, 1998.

[8] Hunt, G. and Brubacher, D. Detours: Binary interception of Win32 functions. Technical Report MSR-TR-98-33, Microsoft Research, 1999.

[9] Lauria, M., Pakin, S., and Chien, A.A. Efficient layering for high speed communication: Fast Message 2.x. In *Proc. of the 7th High Performance Distributed Computing (HPDC7) conf.*, 1998.

[10] Milojicic, D., Zint, W., Dangel, A., and Giese, P. Task migration on the top of the mach microkernel. In *Proc. of the 3rd USENIX Mach Symp.*, pages 273–289, 1993.

[11] Pakin, S., Karamcheti, V., and Chien, A. Fast Message (FM): efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, Vol.5:60–73, 1997.

[12] Paoli, D. and Goscinski, A. The RHODOS Migration Facility. Technical Report TR C95/36, School of Computing and Mathematics, Deakin University, 1995.

[13] Rozier, M., Abrossimov, V., Gien, M., Guillemont, M., Hermann, F., and Kaiser, C. Chorus (Overview of the Chorus distributed operating system). In *Proc. of USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, 1992.