Efficient Switching Supports of Distributed .NET Remoting with Network Processors

Chung-Kai Chen, Yu-Tin Chen, Yu-Hao Chang, Cheng-Wei Chen,
Chih-Chieh Yang and Jenq-Kuen Lee
Department of Computer Science, National Tsing-Hua University, Taiwan
Email:{ckchen, ytchen, yhchang, cwchen, ccyang}@pllab.cs.nthu.edu.tw
jklee@cs.nthu.edu.tw

Abstract

Distributed object-oriented environments have become important platforms for parallel and distributed service frameworks. Among distributed object-oriented software, .NET Remoting provides a language layer of abstractions for performing parallel and distributed computing in .NET environments. In this paper, we present our methodologies in supporting .NET Remoting over meta-clustered environments. We take the advantage of the programmability of network processor to develop the content-based switch for distributing workloads generated from remote invocations in .NET. Our scheduling mechanisms include stateful supports for .NET Remoting services. In addition, we also propose scheduling policy to incorporate workflow models as the models are now incorporated in many of tools of grid architectures. Experiments done at clusters with IXP 1200 network processors show that our scheme can significantly enhance the system throughput (up to 55%) compared to NLB method when the traffic is heavy. Our schemes are effective in supporting the switching of .NET Remoting computations over meta-cluster environments.

I. Introduction

Distributed object-oriented environments have become important platforms for parallel and distributed service frameworks. Among distributed object-oriented software, .NET Remoting provides a framework that allows objects to interact with each other across the boundaries. It separates the remote object from an application boundary and from a specific communication mechanism. By hiding the complexities of calling methods on remote objects, you can build widely distributed applications. In the .NET Remoting Framework, channels are used to transport messages to and from remote objects, and the .NET Remoting infrastructure provides two types of channels that can be used to provide a transport mechanism for the distributed applications - the TCP channel and HTTP channel. TCP channel is a socket-based transport that utilizes the TCP protocol for transporting the serialized message stream across the .NET Remoting boundaries while HTTP channel utilizes the HTTP protocol for transporting the serialized message stream across the Internet and through firewalls. As other networking applications, such as HTTP, server cluster is deployed for serving tremendous request. To leverage the request load among servers and optimize the cluster utilization, it is necessary to apply a load balancing mechanism in server clusters.

Efficient supports for Java remote method invocation have been important topics for investigations, as RMI provides a layer of abstractions for communications. Research results include an open RMI implementation which makes better use of the object-oriented features of Java [14]. ARMI [12] and Manta [9] systems reduce various drawbacks in RMI and to provide new RMI style systems with added functionality. KaRMI [11] is a better way to implement RMI with exploiting Myrinet hardware features to reduce latencies for high performance. Also the broad range of RMI applications was done in [2]. RMI with wireless environment is also done by Lee and Chen in [15], [4]. The Aroma system [10], which is Java-based middleware that aims to exploit the Java RMI to replicate

[1]The work was supported in part by Microsoft .NET resarch grant (2003-2004) and Intel research grant for network processors (2003-2005).

objects for availability and adaptability. The dynamic proxy is used as an interceptor to extend the capability of Java RMI. In addition, specifications for RMI programs over heterogeneous network environments are done in [3]. An abstraction for object interaction in a P2P environment, called query/share (QS) [5], is implemented in Java and relies heavily on the concept of dynamic proxy. Similar to Java RMI, .NET provides remoting as a way for application in different machines/domains to communicate with each other. All the method calls along with the parameters are passed to the remote object through the HTTP channel or TCP channel. With more layers of abstractions, it's an interesting issue to abstract channels and transport layers for supporting heterogeneous network environments via dynamic proxy in .NET environments.

In this paper, we address the issues in supporting .NET Remoting over meta-cluster environments. We take the advantage of the programmability of network processor to develop the content-based switch. Stateful supports for .NET Remoting services are also incorporated. Our work has .NET Remoting applications classified into two separate channels in one application, one is for stateful, and another is for stateless. We then try to dispatch jobs for stateless applications, and also for the scheduling of stateful invocations. In addition, we also incorporate workflow models for tasks to be scheduled into our frameworks. This is due to many of the tools of grid architectures now are with workflow model supports [8]. In the first step of our scheduling policy, we perform scheduling policy for statful jobs in the workflow models. With the initial placements of processor allocations, we then perform the scheduling policy for stateless applications in the second phase. Timeout constraints for stateful tasks are incorporated so that it might roll back processor assignments for stateful tasks during the second phase. This mechanism give load-balancing for stateless tasks while also perform load-balancings of stateful tasks when the timing constraints are met. Our work, to our best knowledge, is the first work to address issues in supporting .Net remoting services for both stateful and stateless methods with network processor supports.

Our experimental platform, Intel IXP1200, contains a StrongARM core of 232 MHz and six programmable 32-bit RISC processors of 232 MHz (a.k.a. microengine). With the benefit of pipeline model, IXP1200 could guarantee wire-speed (up to 622 Mbit/s, OC12) packet processing performance. The whole system implementation is divided into two parts, one is the control system executed in StrongARM core and the other one is data path system executed in microengines. The control system is implemented in ANSI C code; the system feature includes downloading the microcode to microengine, maintaining the related tables in SRAM and SDRAM, and determining the routing path for new .NET Remoting request. The data path system is implemented in microcode, a kind of assembly code designed for microengines of Intel network processor. The functionality of data path system including parsing and rewriting the packet header and deliver the exception packet to StrongARM core. The communication between StrongARM core and microengines was archived by a resource manager and scratch memory. Experiments done on IXP 1200 network processors show that our schemes are effective in supporting .NET Remoting computations over meta-cluster environments.

The rest of this paper is organized as follows. Section II provide an introduction for the basics of .NET Remoting. Next, Section III then presents the frameworks for meta-cluster supports for .NET Remoting with the assistance of IXP network processors. Experimental results is then presented in Section IV and Section V concludes this paper.

II. Technology Basics

A. .NET Remoting Fundamentals

In the .NET Remoting framework, key features includes activation, lifetime management, formatter, and communication channel. .NET Remoting has two types of activation model, server-activated and client-activated. Server-activated type includes singleton and single call modes. In the singleton mode, no more than one instance will be active at any time. It is suited for a stateful programming model since it can maintain state between method calls. In the single-call mode, .NET Remoting infrastructure will activate a new instance for every method invocation. This is a stateless programming model useful for applications such as web service. The lifetime management in .NET Remoting is lease-based. Client-activated objects are under the control of the lease manager that will trigger garbage collection while the lease expires. The lifetime of singletons are also controlled by lease-based lifetime. Formatters are used for encoding and decoding the messages before they are transported by the channel. Channel is responsible for transporting messages across remoting boundaries. Figure 1 illustrates the .NET Remoting architecture and its elements.

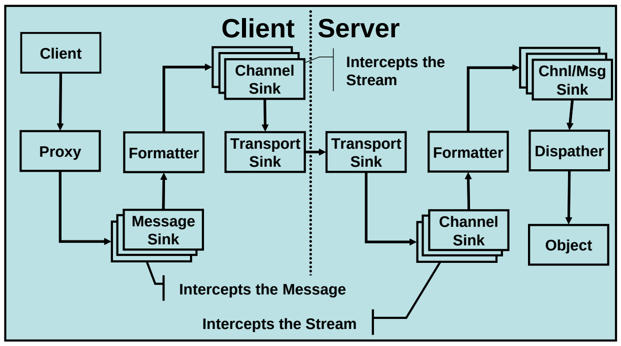


Fig. 1. The .NET Remoting architecture.

When the client makes a remote method call, the client actually makes a call on a proxy object, which in turn converts the stack frame of the method call into a message object. The message object will encapsulate the information related to the method call and will be passed to a set of message sink chains which can do various processing on the message object. The message object will go through the message sink chains until it reaches a necessary sink called formatter sink, which is the first sink in the channel's sink chain and is responsible for serializing the message object into a byte stream according to certain wire format. Then the stream will be passed through several channel sinks for further processing. The last sink in the channel sink chain is the transport chain, which is also a necessary sink and is responsible for transporting the stream over the wire by using a specific transport protocol. The server-side will do almost the reverse processing as the above mentioned. The dispatcher in server-side is indeed a StackBuilderSink which will convert the message object into a stack frame and actually make the method call on the remote object. It also packages the return result and call-by-out arguments into a message object and returns to the client-side. In our work, the network processor will be used to work as a scheduler for both client-activated and sever-activated remoting invocations in .NET environments.

B. Network Processors

Our experimental platform, IXP1200 [20], contains a StrongARM of 232 MHz and six programmable 32-bit RISC processors of 232 MHz (also known as microengine). Microengine was designed as a 5-stage pipeline processor and each also contained four hardware threads. With such parallel processor architecture, IXP1200 could guarantee wire-speed (up to 622 Mbps, OC12) packet processing performance to process .NET remoting packets.

Figure 2 shows the hardware architecture of IXP1200, in addition to rich set of processors, IXP1200 integrates two standard memory interfaces, SRAM and SDRAM, and supports PCI and IX Bus interfaces. The PCI interface allows IXP1200 to connect a host CPU which could take charge of system management and then leave StrongARM as an exception processor; or connect to other PCI devices such as 802.11x wireless card to gain the access capability of wireless network. The IX Bus interface allows IXP1200 to connect networking devices such as Ethernet MACs and ATM SARs or even another IXP1200 processor to deal with higher speed network. IX Bus Unit also integrates a hash unit which supports 48 and 64 bits hashing. Hashing is useful when performing address

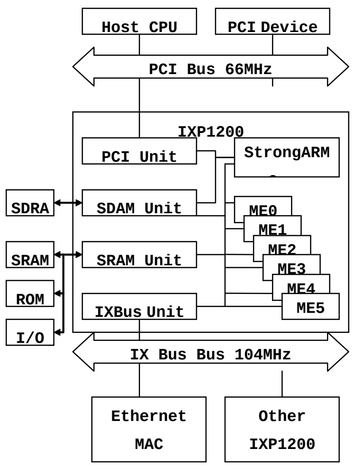


Fig. 2. The IXP 1200 hardware architecture.

lookup especially in network application. These functional units were interconnected by the high speed internal buses. Besides SRAM and SDRAM memory storage, the IXP1200 contains an on-chip 4KByte Scratchpad RAM within IX Bus unit. These three memories vary in latency, capacity, and bandwidth. The SDRAM has the largest capacity and bandwidth but longest latency. On the contrary, the Scratchpad has the smallest capacity and half bandwidth of SDRAM but shortest latency. Programmers could access these three memories concurrently and make use of their characteristics.

III. Efficient Switching Support for .NET Remoting

For meta-cluster supports with .NET remoting, the workload dispatcher is generally needed. Loading balancing mechanism is divided into centralized [6] [1] and distributed [13] versions. We focus on the centralized version in our work. The centralized mode installs a gateway in front of the cluster. The gateway parses incoming request and makes appropriate routing decisions according to specific request attribute (such as source IP address and URL) and server workload feedbacks. The bottleneck for the .NET remoting dispatchers often occurs in the gateway because it needs high computation power to process a huge number of remoting requests. In addition, if the application is stateful, the gateway will consume additional cost to keep the coherence of sessions. We demonstrate how to distribute workloads of .NET remoting with the assistance of IXP 1200 network processors.

A. Remoting Switch

We take advantage of the programmability of network processor to develop the content-based switch. Figure 3 shows the system architecture of our design. The network processor `NP` serves as the gateway of remoting services hosted on each backend servers. All TCP channel connections of remoting going to the servers are brokered by the network processor. It uses its special hardware architecture to do fast TCP/IP header rewriting for directing packets back and forth. A TCP connection table is maintained in the memory space of the network processor to keep track of the connection information. It includes the IP and port information of the client and the connected server for each connection.

As a gateway of the backend servers, the job of `NP` is to dispatch remoting invocations concerning the load-balancing issues and the session semantics. For stateless remoting services, `NP` chooses a least load server to dispatch invocations; for stateful remoting services, `NP` has to make sure that invocations belonging to the same session will be dispatched to the same server. In Figure 3, `RO1`, `RO2` and `RO3` are all remoting objects that contain the intended operations for remote invocations.

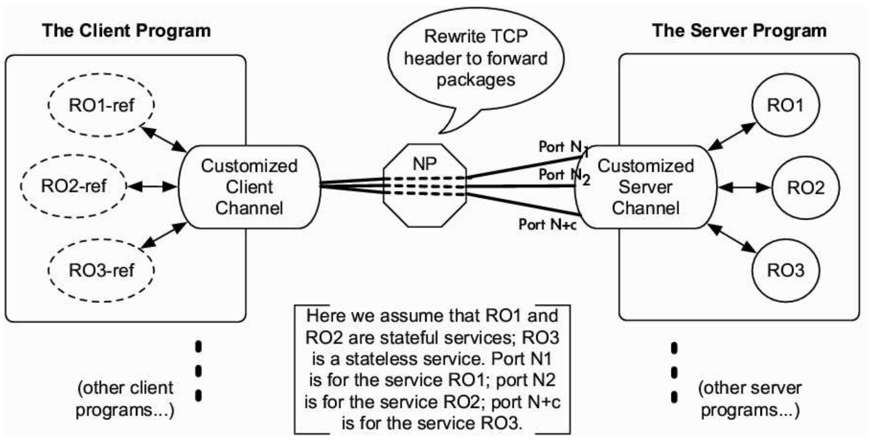

The Client Program
Rewrite TCP header to forward packages
The Server Program
RO1-ref
RO2-ref
RO3-ref
Customized Client Channel
NP
Port N1
Port N2
Port N+c
Customized Server Channel
RO1
RO2
RO3
(other client programs...)
Here we assume that RO1 and RO2 are stateful services; RO3 is a stateless service. Port N1 is for the service RO1; port N2 is for the service RO2; port N+c is for the service RO3.
(other server programs...)

Fig. 3. The system architecture of using the network processor as the remoting service gateway.

The `RO1-ref`, `RO2-ref` and `RO3-ref` in the client side are the TransparentProxy objects referring to `RO1`, `RO2` and `RO3` respectively. Both the proxy object and the remoting object use a channel object to manage network connections for data transportation. In this system, we design and deploy a pair of extended channel objects to automatically distribute remoting invocations into different TCP connection ports according to their service types. By doing this, `NP` can identify the service types through the examination of the destination port of incoming request packets. About the distribution of services on different ports, we use a map data structure to record the assigned port for each remoting service. Especially, all stateless services are bound to the port number large than `c`, where `c` is a selected constant. This map information can be a part of the remoting service deployment configurations and is accessible by the clients and the servers. We describe the distribution mechanism done by the channel objects below.

- **Client Channel Object** When the `SyncProcessMessage` or `AsyncProcessMessage` method of the client channel object is called in order to start a remoting invocation, it analyzes the parameter `IMessage` object to fetch the remoting service name. The mapped port for that remoting service is looked up by the map and is used for sending request packets.
- **Server Channel Object** When the server channel object is first instanciated, it looks up the map for all the currently used ports for remoting services. Then it opens corresponding server sockets on these ports to listen to connections.

Algorithm 1 shown in Figure 5 gives the detailed dispatching process done by the network processor. Figure 4 also depicts the control flow of this dispatching algorithm that helps the understanding of Algorithm 1. The main effort of this dispatching algorithm is to decide which server a TCP connection is going to be connected with. It is the place where dispatching decisions are made. Once a server is chosen and the connection is constructed, all remoting invocations go through this link are served by this server. Here we can have the channel objects periodically discard connections in purpose for the reconstruction of connections to less load servers. The network processor records the IP and port information of the client and the selected server in the TCP connection table called `TCT` for each constructed connection. The remoting request packets with the same source IP, the same source port, and the same destination port will be directed to the same destination IP according to `TCT`. The response packets from the servers are also directed to the correct clients by this connection table. Notice that the destination port mentioned here is used to identify remoting services since we have distributed different services to go on different ports in our customized client channel object. Step 3 of Algorithm 1 does the checking to see if the incoming packet is already in a `TCT` entry. In addition, with the destination port plus the IP and port information of the client, we can construct a session

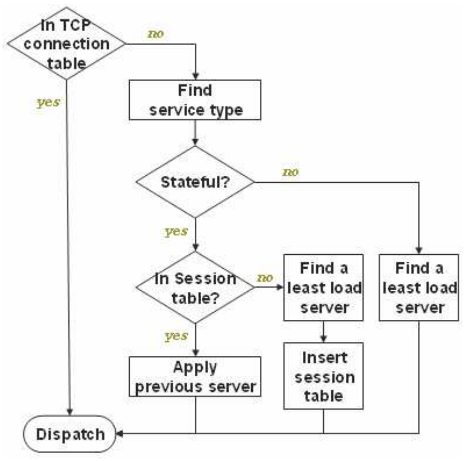


Fig. 4. The flow chart of the dispatching process.

table ST. ST is then used to track existing sessions for stateful services. If the network processor finds that no TCP connection exists for the incoming packet and the destination port shows that it belongs to a stateful service, it will then look up ST to find out the previous assigned server for this service. Step 4 checks if the incomong packet is in the range of stateful services. In addition, State 5 and state 6 checks if the incoming packet is a singleton or a client-activated method. Note that there are three kinds of methods in .Net remoting. Single-call is stateless, and stateful methods include singleton and client-activated methods. Our scheduling policy fully supports these three semanitcs for .Net methods. A time field is also kept in ST to determine the expiration of a session. Step 10 does the checking for the expiration of a session. The network processor can also invalidate the content in TCT and ST on purpose in order to reallocate stateful services to new servers for load-balancing issues, the Remoting proxy in the client side will detect a network failure exception and then can try to construct a new TCP connection to the backend. Step 7 directs the connection construction request to least load servers. Once the least server is found, step 8 checks if the incoming method is indeed a stateful request. A ST will be created if this is a new stateful request. Finally, the algorithm does TCP header rewriting to forward packets to and from the intended server of that connection. This is done between step 11 and step 13.

B. Programming Network Processors

The whole system implementation is divided into two parts, one is the control system executed in StrongARM core and the other one is the data path system executed in microengines. The control system is implemented in ANSI C code; its feature includes downloading the microcode to microengines, maintaining the related tables in SRAM and SDRAM, and determining the routing path for new remoting request. Figure 6 gives a code segment for rewriting the packet header. The functions have prefix started with “ix_” are SDK library provided by Intel. Variables “iphdr” and “tchhdr” are pointers to ip header and tcp header of a packet, respectively. The data path system is implemented in microcode, a kind of assembly code designed for microengines of IXP 1200. The functionality of a data path system includes parsing and rewriting the packet header and delivering the exception packet to StrongARM core. The communication between StrongARM core and microengines is achieved by a resource manager and scratch memory. Figure 7 gives a code segment for extracting ip and tcp header. “.local” is a directive to declare register for later using. Macro “xbuf_extract” extract a numeric byte field from transfer register buffer, “$$ip_header”, to general-purpose register.

Algorithm 1: **A dispatching algorithm for handling stateless and stateful Remoting invocations by using dedicated TCP channel connections.**

In the context TCT stands for the TCP connection table that is maintained to track existing TCP connections. Each row in TCT contains four columns: the source IP, the source port, the destination IP and the destination port of one TCP connection. The destination port of each TCP connection is restricted to be within one of the three integer ranges $R_{single-call}$, $R_{singleton}$ and $R_{client-activated}$. They are used to identify the connections dedicated to single-call, singleton or client-activated Remoting invocations respectively. Another table ST is the session table maintained to track existing sessions of stateful services. Each row in ST contains five columns to keep the information about one session. They are the source IP, the source port, the destination IP, the destination port and the access time of this session.

```
Begin
Step 1. Receive a packet from the clients.
Step 2. Read the source IP information in the TCP/IP header into $SrcIP$.
        Read the source port information in the TCP/IP header into $SrcPort$.
        Read the destination port information in the TCP/IP header into $DestPort$.
Step 3. if (there exists a row $Conn$ in $TCT$ that (the source IP of $Conn$ == $SrcIP$)
        && (the source port of $Conn$ == $SrcPort$) && (the destination port of $Conn$
        == $DestPort$))
            Read the destination IP column of $Conn$ into $DestIP$.
            Goto Step 13.
Step 4. if ($DestPort$ is in range $R_{single-call}$)
            Goto Step 7.
Step 5. if ($DestPort$ is in range $R_{singleton}$)
            if (there exists a row $Sess$ in $ST$ that (the destination port of $Sess$ ==
            $DestPort$)
                Goto Step 10.
Step 6. if ($DestPort$ is in range $R_{client-activated}$)
            if (there exists a row $Sess$ in $ST$ that (the source IP of $Sess$ == $SrcIP$) &&
            (the source port of $Sess$ == $SrcPort$) && (the destination port of $Sess$ ==
            $DestPort$))
                Goto Step 10.
Step 7. Find the least load server and write its IP to $S$.
Step 8. if ($DestPort$ is in range $R_{single-call}$)
            Goto Step 12.
Step 9. Create a new row $Sess'$.
            Assign $SrcIP$ to the source IP column of $Sess'$.
            Assign $SrcPort$ to the source Port column of $Sess'$.
            Assign $DestPort$ to the destination port column of $Sess'$.
            Assign $S$ to the destination IP column of $Sess'$.
            Assign the current time to the time column of $Sess'$.
        Insert $Sess'$ to $ST$
        Goto Step 12.
Step 10. if ((the current time) - (the time column of $Sess$) $>=$ $SESSION\_TIMEOUT$)
            Delete $Sess$ from $ST$.
            Goto Step 7.
Step 11. Read the destination IP column of $Sess$ into $S$.
Step 12. Assign $S$ to $DestIP$.
Step 13. Rewrite the TCP/IP header of the packet with $DestIP$ as the destination IP.
Step 14. Send out the packet.
End
```

Fig. 5. The dispatching algorithm for remoting invocations.

```
ix_tcphdr_src_port_write(tcphdr, current->ip_dport);
ix_tcphdr_dest_port_write(tcphdr, current->ip_sport);
ix_tcphdr_seq_write(tcphdr, seq));
ix_tcphdr_ack_write(tcphdr,ack));
ix_tcphdr_flags_write(tcphdr, ACK_SYN_MASK));
ix_checksum_calc_segment_checksum(iphdr, (void*)tcphdr, &chksum,1);
```

Fig. 6. A C programs for rewriting packet header.

```
.local nsrc ndst nsport ndport chksum_delta seq
    xbuf_extract(nsrc, $$ip_header, 0, NSIP)
    xbuf_extract(ndst, $$ip_header, 0, NDIP)
    xbuf_extract(nsport, $$ip_header, 0, NSPORT)
    xbuf_extract(ndport, $$ip_header, 0, NDPORT)
    xbuf_extract(chksum_delta, $$ip_header, 0, DT)
    xbuf_extract(seq, $$ip_header, 0, SEQ)
.endlocal
```

Fig. 7. The Microcodes for parsing TCP packet header.

IV. Load Balancing Mechanisms

The key scheduling policy for our algorithm in handling both stateful and stateless services of .NET remoting are shown earlier in Algorithm 1. Step 7 of this dispatching algorithm is to find the least load server for dispatching. Different scheduling methods can be plugged in for this step. In the following, we propose two methods for this purpose. The first method is to schedule tasks to the server minimizing the estimated task time. The second method incorporates workflow models for task scheduling. This is to exploit the fact that many of the tools of grid architectures are now equipped with workflow model supports [8].

A. ETT Scheduling Methods

According to the characteristics of the applications, we propose an algorithm which dispatches the request by the difference of the cpu computing power and the network bandwidth. The ***Estimated Task Time*** (ETT) model is defined as

$$ETT(n_i, s_j) = \frac{cc(n_i)}{P_j \times (1 - CPU\_load)} + \frac{d(n_i)}{W_j \times (1 - bandwidth\_load)} \tag{1}$$

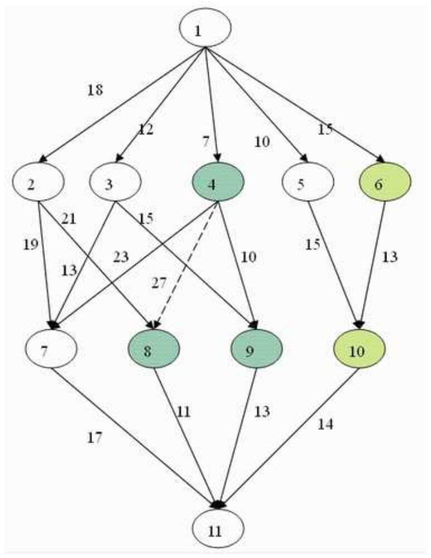
The **cc**(n$_i$) is the cycle of task n$_i$, the **d**(n$_i$) is the amount of data need to be communicated. **P**$_j$ is the clock rate of the processor in server s$_j$. **W**$_j$ is the network bandwidth of server s$_j$. Note that cpu load and network loadings can be gotten from feedbacks of the back-end computing servers, periodically. The characteristics of jobs such as computing time and communication time can also be gotten by profiling schemes of systems. The scheduling algorithm for Step 7 of Algorithm 1. is now given in Figure 8.

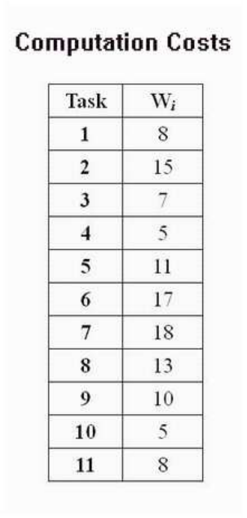
B. Scheduling Methods with Workflow Graphs

We now present a scheduling method which incorporates workflow models for task scheduling. A workflow of tasks is represented as directed acyclic graph (DAG). An example of such a graph is shown in Figure 9. Nodes represent application tasks and edges represent data communication. The computation costs and communication costs are stored in a $n \times 1$ and $n \times n$ matrix, respectively. In the example graph, tasks $n_4, n_6, n_8, n_9, n_{10}$ are stateful tasks associate with two different services. The graph also comes with information to mark the stateful tasks when the timeout constraint for

```
Find_Least_Load_Server(){
    while there is a incoming request $n_i$ do
        for each server $s_j$ do
            Compute ETT($n_i$, $s_j$)
        Assign request $n_i$ to the server $s_k$ that minimizes ETT of request $n_i$
    end while
    }
```

Fig. 8. The ETT Load-Balancing Algorithm.





Computation Costs

Task	W_i
1	8
2	15
3	7
4	5
5	11
6	17
7	18
8	13
9	10
10	5
11	8

Fig. 9. An example of a task graph with 11 tasks.

expiration is raised. In this case, the stateful tasks as the successors can be redirected to other server for load balancing. This timeout information is presented as the dotted line of the edge. In our example graph, the edge between tasks n_4 and n_8 is with timeout edge. We assume every server can execute maximum k tasks in parallel. Tasks will be queued until the running tasks are less than k in a server and the computation cost will be n times of the original execution time of a task when there are n tasks executed on a server.

We have defined several attributes for task scheduling. The rank of the tasks represent the priorities of the scheduling order. The $rank(n_i)$ is the approximation of the length of the longest path from the task n_i to the exit task. The rank of task n_i is defined by

$$rank(n_i) = w_i + \max_{n_j \in succ(n_i)} (c_{i,j} + rank(n_j)), \qquad (2)$$

where w_i is the computation cost of task n_i, $succ(n_i)$ is the set of the immediate successors of task n_i, $c_{i,j}$ is the communication cost of edge(i, j). According to the rank, we schedule tasks by decreasing order of rank.

Our scheduling algorithm presents a two-phase scheduling policy. In the first phase, we perform a pre-scheduling for all stateful tasks, and then we perform scheduling for stateless tasks in the second phase. In our first phase, we first mark all the stateful tasks by traversing the graph. If the edge before the task is marked as timeout, then all the tasks following the edge will be recognized as a new stateful task group. After separating the stateful tasks into different groups, we can then schedule each group one by one. We use the following equation to estimate load of the stateful tasks which has been scheduled to the server.

$$Load(s_i) = \sum_{\forall g_j \ has \ been \ scheduled \ to \ s_i} \{ \sum_{\forall n_k \in g_j} R_k \}, \tag{3}$$

where s_i is the i-th server, R$_k$ is the remaining computation time of task n_k, and g_j is the j-th group of the stateful task groups. We also have

$$AddLoad(s_i, g_t) = Load(s_i) + \{ \sum_{\forall n_k \in g_t} R_t \}. \tag{4}$$

In order to balance the group load of the stateful tasks, we use the AddLoad function to calculate the total computation cost of each group when adding a new scheduled group. We then dispatch them to servers by picking up the minimum one. The scheduling algorithm is illustrated in Figure 10.

After all the stateful tasks has been scheduled, we subsequently schedule the stateless tasks by the order generated by rank. The phase2_stateless_scheduler routine in Figure 10 presents the algorithm for the second phase of the scheduling. When a stateful task leaves the queue and prepare to be executed, we check the timeout value of the stateful group which was separated by the given timeout mark. To see the timeout will happen or not, if not, we will redirect the rest stateful tasks to the original server to keep the correctness of the stateful service. In this case, we also indicate the roll-back of the scheduling results for stateful tasks, and re-run the stateful scheduler in the phase one for the remaining stateful tasks. For a stateless task, we use the following function to estimate the finish time of the stateless task executing on the servers.

$$EFT(n_i, s_j) = Exec(w_i, avail[s_j], k) + \max_{n_m \in pred(n_i)} (AFT(n_m) + c_{m,i}), \tag{5}$$

where $pred(n_i)$ is the set of immediate predecessor tasks of task n_i, and $avail[s_j]$ is the earliest time at which server s_j is ready for task execution. AFT(n_m) is the actual finish time of the task n_m. $Exec(w_i, avail[s_j], k)$ is the execution cost of task n_i with computation cost w_i execute on the server s_j which can parallel execute at most k tasks from time $avail[s_j]$. And we choose the server with the minimum EFT to schedule. The last paragraph of the second routine illustrates this idea.

Now we use the algorithm to schedule our sample graph. We assume each server can execute two tasks in parallel, and there are three servers. According to the first phase, we need to schedule the stateful tasks by equation 3. We traverse the graph to find out the stateful tasks and separate them into groups, note that there is a timeout mark between task n_4 and n_8. We therefore can separate them into three groups which are $g_1 = \{n_4, n_9\}$, $g_2 = \{n_6, n_{10}\}$, and $g_3 = \{n_8\}$, then schedule g_1, g_2, and g_3 to server s_1, s_2, and s_3 respectively by equation 3. Once the stateful tasks has been scheduled, the rank of each task need to be calculated to decide the scheduling order. Using equation 2, the rank values of tasks are $\{n_1 = 93, n_2 = 77, n_3 = 63, n_4 = 71, n_5 = 53, n_6 = 57, n_7 = 43, n_8 = 32, n_9 = 31, n_{10} = 27, n_{11} = 8\}$. Then we sort the rank by decreasing order to get the scheduling order, which is $\{n_1, n_2, n_4, n_3, n_6, n_5, n_7, n_8, n_9, n_{10}, n_{11}\}$. Once the scheduling order is obtained, the tasks can be scheduled subsequently. By choosing the task with highest rank, task n_1 will be scheduled first. We use the EFT (equation 5) to find out the minimized execution time on servers. Because task n_1 has no predecessor, the result of EFTs is equal to 8: ($\forall s_i$, $EFT(n_1, s_i) = Exec(8, avail[s_i], 2) + 0 = 8$, the avail[$s_i$,2] is 0, because the task n_1 is the first task of each server.) By the scheduling order, the next task n_2 is also a stateful task whose EFTs need to be calculated: ($EFT(n_2, s_1)$= $Exec(15, avail[s_1], 2)$ + $max\{AFT(n_1)$ + $c_{1,2}\}$ = 15 + (8+0) = 23; $EFT(n_2, s_2)$ = 15 + (8+18) = 41; and $EFT(n_2, s_3)$ = 15 + (8+18) = 41.) From such results, we can schedule the task n_2 to server s_1 which has the minimum EFT value. The task n_4 is the next task need to be scheduled by the order. Since the task n_4 is a stateful task, we dispatch task n_4 to the server s_1 which was decided previously. According to the algorithm, we can dispatch the rest tasks and get a simulated result of the sample graph as shown in Figure 11. The timeout of task n_8 will not happen

```
Input: A task graph G with the computation cost, communication costs,and the stateful groups.

Let K = the number of servers.
Let P = the number of tasks can be executed in parallel.

Phase1_Stateful_Scheduler(){
    while there is a unscheduled group $g_i$ do
        for each server $s_j$ do
            Compute the AddLoad($s_j$, $g_i$).
        Assign the tasks of $g_i$ to the server $s_k$ that minimizes AddLoad($s_k$,$g_i$).
    end while
}
Phase2_Stateless_Scheduler(){
    while there is a un-scheduled task in the graph do
        Find the highest ranked task among un-scheduled tasks, say $n_i$, for scheduling
        if task $n_i$ is stateful {
            if (the timeout constraint for expiration is raised for task $n_i$)
                  and ((the current time) - (the time for last done task of this group))< TIMEOUT {
                Assign the tasks of the group to the server which the task of this group has been scheduled.
                Revise this scheduling information to call Phase1_statefull_Scheduler() to re-do remaining stateful tasks.
            }
            else do
                Assign the stateful request to the server assigned at phase1
                  and update the session table.
            end if
        }
        else {/* Schedule stateless tasks */
            for each server $s_j$ do
                Compute EFT($n_i$, $s_j$).
            Assign request $n_i$ to the server $s_k$ that minimizes EFT of request $n_i$.
        }
        Update the connection table
    end while
}
```

Fig. 10. Load-Balancing algorithm for the application with a workflow graph.

when the application is executed, it should be redirected to the server s_1 to keep the stateful tasks correctness. The final schedule length of the sample graph is 86.

V. Experiments

The experimental environment includes two clients and two InfiniBand clusters. These two clients each belongs to individual subnet issue remoting requests to the load balancer. On the other hand, two identical clusters composed of two P4-2GHz servers were employed to run Remoting server applications. In the hardware configuration of NLB experiment, we replace IXP1200 with a 4-port 100Mbps switch such that the maximum aggregate throughput from servers could reach to 200Mbps. The throughput is given in Figure 12 where we compare the throughputs of our proposed dispatcher for .NET remoting with that of Microsoft Network Load Balancing (NLB) technology. NLB is a distributed methodology in which each server in the NLB cluster will receive the same copy of packet. Our work is to have .NET remoting framework classified into two separate channels in one application, one is for stateful, and another is for stateless. We then try to dispatch jobs by using the proposed mechanism in Section 4.2.

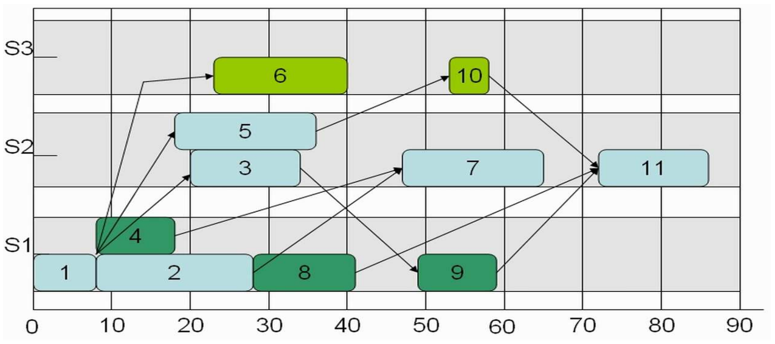


Fig. 11. The result of the sample application using our scheduling algorithm.

To measure the throughput, two clients concurrently called a method transferData() which will transmit a piece of data buffer to the server and receive another data buffer back. The ratio between transmitting and receiving buffer size is adjustable. There are totally 128Kbytes exchanged in one method call, that is, a client transmits 128*ratio Kbytes to a server and receives 128*(1-ratio) Kbytes each time transferData() was called. We adjusted the ratio from 0 to 1 progressively and measured the aggregate throughput of two clients. Figure 12 shows the experimental result.

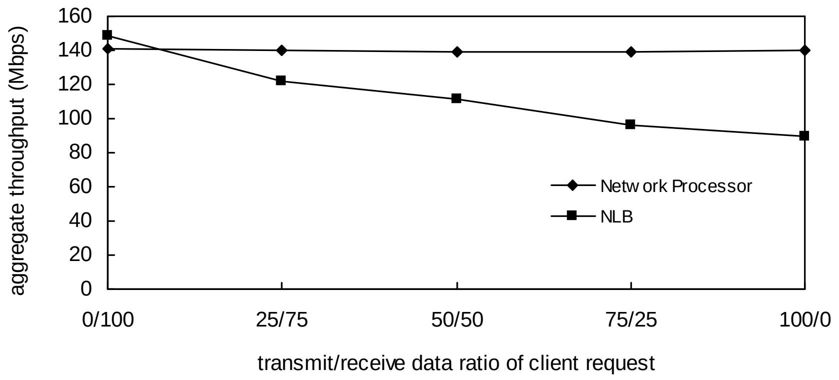


Fig. 12. The throughputs with two different load-balancing mechanisms.

When only the client received data from the server (ratio = 0), the performance of NLB is close to 150Mbps which is 5.4% better than our system. However, the throughput of NLB declined as the ratio increased while our system remained high throughput. When the ratio was raised to 0.08, both two systems have similar performance. The throughput of NLB decreased to 90Mbps when only clients transmitted data to servers. This is because NLB is a distributed methodology in which each server in the NLB cluster will receive the same copy of packet. The NLB driver in front of TCP/IP protocol stack will decide to forward the packet to protocol stack or discard it according to given packet information (ex. TCP/IP header). The aggregate packet received on server side equals the amount of packets transmitted from client side multiplied by the server number of NLB cluster. Consequently, NLB may waste network bandwidth while our system does not.

Next, we compare the scheduling effects with our scheduling policy. In the experiment, we developed a Remoting method getPrimeCount(int X) that could calculate the count of prime number smaller than integer X. (Note that the CPU consumption time of this method is positively proportional to X.) On the client side, the client program invoked the Remoting method repeatedly by an interval time Y (milliseconds). To simulate real environment, we generated Y by an exponential function floor(-

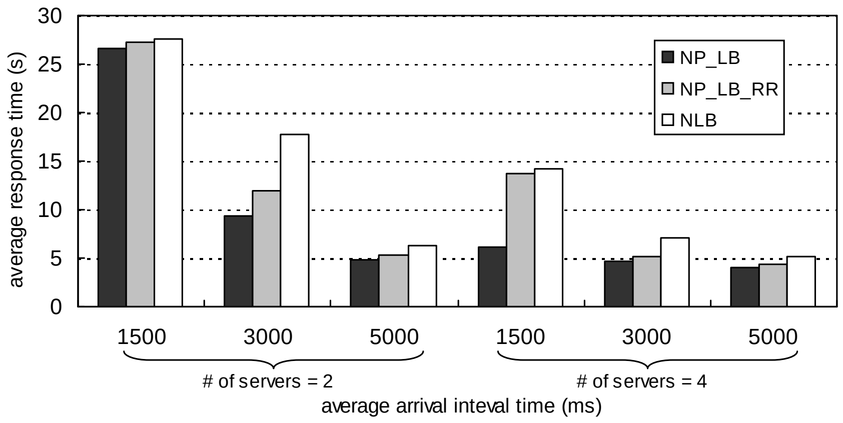


Fig. 13. The response time among three load balancing mechanisms with interval time (1500, 3000, 5000).

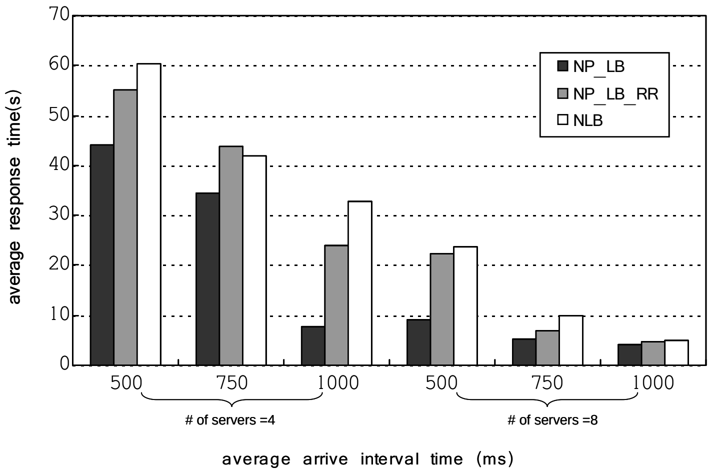


Fig. 14. The response time among three load balancing mechanisms with interval time (500, 750, 1000).

log(rand(0,1))* intervalMulti). Similarly, the number X which the client wants to calculate was also generated by an exponential function floor(-log(rand(0,1))* primeRangeMulti). Therefore, the average X and Y are primeRangeMulti and intervalMulti, respectively. According to primeRangeMulti=30K and different intervalMulti values (1500, 3000, and 6000 ms) for (2, 4) nodes and (500, 750 and 1000 ms) for (4, 8) nodes, we generated 3 X-Y distribution samples each with 1000 points. It means the client program will use these samples to call getPrimeCount(X) 1000 times, totally. We measured the response time of each method call in three load balancing mechanisms. The first one is our load balancer (called NP_LB), the second one is also our system but it dispatches jobs in round-robin fashion instead of dispatching by servers' load (called NP_LB_RR). The final one is NLB technology. The results were shown in Figure 13 and Figure 14. We could find out that the average response time of NP_LB is better than NP_LB_RR and NLB in all combinations. When server number is 2 and average arrival interval time is 3 seconds, NP_LB could reduce 47.6% response latency than NLB. Moreover, NP_LB could even reduce 76.1% overhead than NLB when server number is 4 and average arrival interval time is 1 seconds. That is because NP_LB dispatches the request according to servers' load rather than the random selection used by NLB or round-robin selection in NB_LB_RR such that it could get better CPU utilization.

In the following we measure the performance factors in microengine allocations of network processors in our implementation. In the experiment, we try different microengine allocation to examine how it affects load balancer. Just like the previous experiment, we measure the aggregate throughput of two clients which invoke transferData() concurrently. There are 6 combinations listed in Table 15. Except for the first combination, the microblocks for layer-2 bridging processing, load balancing processing, and layer-3 forwarding processing (called main processing for short) was allocated together. Take the third combination (1+4+1) as example; it allocates one microengine for packet ingress, one for packet engress, and the other four for the main processing. The result was shown in Figure 16, combination 1

	ID	Combination
1	1+1	(Ingress+L2+LB+L3) + (Egress)
2	1+1+1	(Ingress) + (L2+LB+L3) + (Egress)
3	1+4+1	(Ingress) + (L2+LB+L3)*4 + (Egress)
4	1+3+2	(Ingress) + (L2+LB+L3)*3 + (Egress)*2
5	2+3+1	(Ingress)*2 + (L2+LB+L3)*3 + (Egress)
6	2+2+2	(Ingress)*2 + (L2+LB+L3)*2 + (Egress)*2

Fig. 15. The effects of microengine allocations.

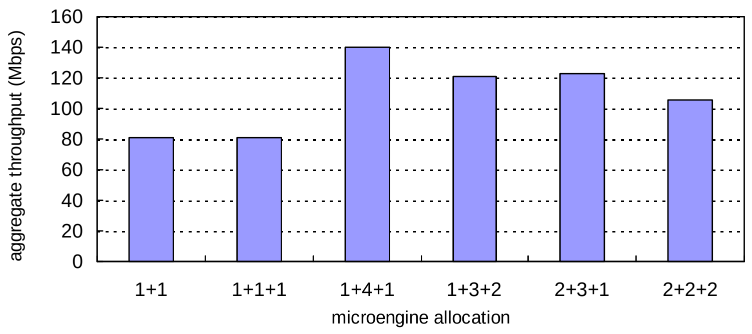


Fig. 16. Throughputs with different microengine allocations.

and 2 have the similar performance although the latter activated one more microengine. It explains that the internal packet forwarding from one microengine to another increase the memory access overhead and therefore eliminates the gain of additional microengine computation power. Combination 3 has the best performance because main processing requires most computation power. It was illustrated by combination 4, 5, and 6. We arranged one microengine from main processing to transmit packets in combination 4. Similarly, two microengines were assigned to receive packets in combination 5. These two groups both result about 14% throughput drop compared with the best one. Nevertheless, the last combination which rearranged two microengine for packet ingress, two for main processing and two for packet egress diminish almost quarter throughput. We could summarize that the quantity of computation power for main processing affect the system performance rather than the computation power needed for packet input and output.

Finally, we also have some simulation results on our workload graph-awared scheduling algorithms. In this experiment, we simulate the network processor scheduling behavior by a program called the dispatcher. The back-end machines are two GHz-level PC connected with 100-baseT Ethernet. The tasks are generated according to a graph generator and are passed to the dispatcher. The dispatcher then fires appropriate Remoting invocations to the corresponding remote objects at the back-end servers. The remote objects simulate the task execution at the servers considering the effects of running multiple tasks on the same machine and the task communication between servers. Te experiment our scheduling algorithms, the graph generator takes several parameters such as the total task number in a graph, the communication-to-computation rate (CCR), the density of stateful tasks, the number of stateful task groups, and a shape value that can control the width and height of the graph.

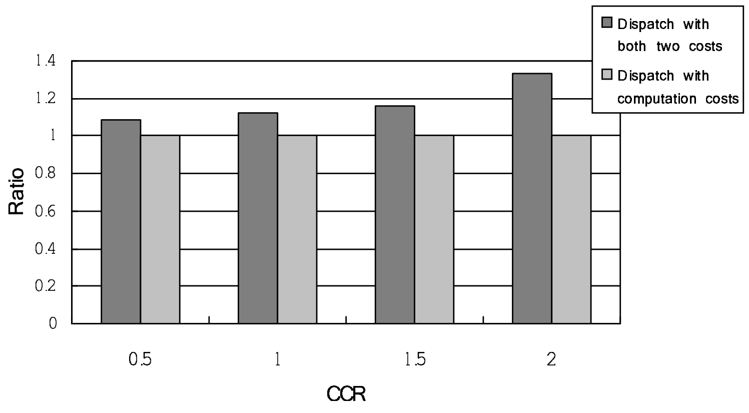


Fig. 17. The throughput ratios of the two dispatchers.

In our scenario, we compare the performance results of two dispatchers with a variety of CCRs. The values of CCR in the simulation are 0.5, 1.0, 1.5, and 2.0, while other parameters are fixed; the number of tasks, the number of stateful group, and the density of the stateful tasks are set to 20, 2, and 0.3 respectively.

There are two dispatchers: the first one takes account of both the communication and computation costs; while the second one takes account of communication cost except computation cost. As Figure 17 shows, the first one achieves more throughput than the second one. And when the ratio of communication cost to computation cost (CCR) increases, the throughput is also improved. The results show the algorithm presented in Figure 10 to be useful for applications with workflows.

VI. Conclusion

In this paper, we presented our methodologies in supporting .NET Remoting over meta-clustered environments. Both Stateful and stateless supports for .NET Remoting services are incorporated. Experiments show that our scheme can significantly enhance the system throughput (up to 55%) compared to NLB method when the traffic is heavy. Our work gave a comprehensive study for efficient support of .NET remoting in the presence of advanced network architectures such as IXP network processors. Our proposed scheduling methods include schemes with or without workflow information of tasks. Further efforts to integrate our scheduling policy with CCA grid environments will be important directions for future research explorations.

References

[1] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, and Debanjan Saha, Design, Implementation and Performance of a Content-Based Switch, in *Proceedings of IEEE Infocom 2000*, Mar. 2000.

[2] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, September–November 1998.

[3] Chung-Kai Chen, Cheng-Wei Chen, Jenq Kuen Lee. Specification and Architecture Supports for Component Adaptations on Distributed Environments, *Proceedings of the IPDPS Conference*, Santa Fe, April 2004.

[4] Cheng-Wei Chen, Chung-Kai Chen, Jyh-Cheng Chen, Chien-Tan Ko, Jenq-Kuen Lee, Hong-Wei Lin, Wang-Jer Wu. Efficient Support of Java RMI over Heterogeneous Wireless Networks, *Proceedings of International Conference on Communications (ICC)*, Paris, June 2004.

[5] Patrick Eugster and Sébastien Baehni Abstracting Remote Object Interaction in a Peer-2-Peer Environment. In *Proceedings of Joint ACM Java Grande - ISCOPE 2002 Conference*, pp. 46–55, 2002.

[6] Robert Haas, Lukas Kencl, Andreas Kind, Bernard Metzler, Roman Pletka, Marcel Waldvogel, Laurent Frelechoux, and Patrick Droz, IBM Research Clark Jeffries, IBM Corporation, Creating Advanced Functions on Network Processors: Experience and Perspectives, *IEEE Network*, July/August 2003.

[7] U. Kremer, J. Hicks and J. Rehg. A Compilation Framework for Power and Energy Management on Mobile Computers. In *Proceedings of the 14th International Workshop on Parallel Computing (LCPC)*, August 2001.

[8] Sriram Krishnan, and Dennis Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proceedings of International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2004.

[9] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java remote method invocation. In *Proceedings of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[10] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Interception in the Aroma System. *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, CA (June 3–4, 2000), pp 107–115

[11] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A More Efficient RMI for Java. In *Proceedings of ACM 1999 Java Grande Conference*, pp. 152–157, June 1999.

[12] R. Raje, J. Williams, and M. Boyles. An asynchronous remote method invocation mechanism for Java, *Concurrency: Practice and Experience*, Vol. 9(11), 1207-1211, 1997.

[13] G. Teodoro, T. Tavares, B. Coutinho, W. Meira Jr., and D. Guedes, Load Balancing on Stateful Clustered Web Servers, in *15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, November, 2003.

[14] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective Remote Method Invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, September–November 1998.

[15] P. C. Wey, J. S. Chen, Cheng-Wei Chen, Jenq-Kuen Lee, Support and Optimization of Java RMI over Bluetooth Environments. In *Proceedings of Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Nov. 2002.

[16] Infiniband Trade Association, Infiniband Architecture Specification, Release 1.0, http://www.infinibandta.org.

[17] UC Berkeley Millennium. Virtual Interface Architecture.
http://www.cs.berkeley.edu/philipb/via/.

[18] Compaq, Microsoft, and Intel, Virtual Architecture Specification Version 1.0, Technical report, Compaq, Microsoft, and Intel, December 1997.

[19] Active IETF Working Groups, internet-drafts, IP over Infini- Band (IPoIB) Architecture, http://www.ietf.org/internet-drafts/draftietf-ipoib-architecture-03.txt.

[20] Intel IXP1200 Network Processor Hardware Reference Manual.