An Empirical Comparison of Java Remote Communication Primitives for Intra-Node Data Transmission

Philip F. Burdette, William F. Jones,
Brian C. Blose, Gregory M. Kapfhammer
Allegheny College
Department of Computer Science

ABSTRACT

This paper presents a benchmarking suite that measures the performance of using sockets and eXtensible Markup Language remote procedure calls (XML-RPC) to exchange intra-node messages between Java virtual machines (JVMs). The paper also reports on an empirical study comparing sockets and XML-RPC with response time measurements from timers that use both operating system tools and Java language instrumentation. By leveraging packet filters inside the GNU/Linux kernel, the benchmark suite also calculates network resource consumption. Moreover, the framework interprets the response time results in light of memory subsystem metrics characterizing the behavior of the JVM. The empirical findings indicate that sockets perform better when transmitting small to very large objects, while XML-RPC exhibits lower response time than sockets with extremely large bulk data transfers. The experiments reveal trade-offs in performance and thus represent the first step towards determining if Java remote communication primitives can support the efficient exchange of intra-node messages.

1. INTRODUCTION

Waldo et al. describe a category of "local-remote" object-based systems where objects are in different address spaces but are guaranteed to be on the same computational node [34]. Initially exemplified by systems such as Spring [27] and Clouds [8], these local-remote systems have become increasingly prevalent, thus highlighting the need for efficient intra-node communication primitives. With the emergence of powerful multi-core processors [24, 30], it may be desirable to construct an efficient local-remote system from the servers that previously executed in a distributed fashion. In an attempt to develop a highly reliable operating system, a local-remote approach could leverage many user-mode servers that support self-repair operations [15, 16]. Moreover, efficient run-time debugging, profiling, and instrumentation techniques often perform intra-node communication with the executing program [6, 35]. While these types of local-remote systems are often useful, flexible, and reliable, they can incur increases in source code complexity and implementation effort due to the use of custom intra-node communication primitives [15, 32, 33].

In contrast to specialized implementations of local-remote communication, the Java programming language furnishes a wide variety of easy-to-program remote communication primitives (RCPs) that have different performance characteristics and functionality. Yet, the Java implementation of two representative RCPs, sockets and eXtensible Markup Language remote procedure call (XML-RPC), were not specifically designed to support communication between Java virtual machines (JVMs) on the same computational node. While previous empirical studies suggest that sockets may perform up to an order of magnitude faster than XML-PRC when transferring a significant amount of data across nodes [2], there is a relative dearth of information about how these primitives support intra-node communication.

To this end, this paper describes a benchmarking framework that uses both Java language and operating system timers, kernel packet filters, and JVM behavior monitors to respectively characterize the response time, network resource consumption, and memory subsystem activity of both sockets and XML-RPC. In particular, the benchmarks support the performance evaluation of Java-based software systems that perform intra-node communication, as depicted in Figure 1. The experimental results indicate that the combined use of simple benchmarks and statistical analysis techniques can identify important trade-offs in the intra-node communication performance of Java programs.

As such, this paper frames and begins to answer the question *can Java RCPs support intra-node communication*? We anticipate that the use of these benchmarks will develop a deeper understanding of Java's remote communication primitives and subsequently lead to decreases in the message passing overhead, code complexity, and implementation effort associated with future local-remote systems that are developed in the Java programming language. In summary, the important contributions of this paper include:

1. A benchmarking framework that evaluates the performance of Java remote communication primitives and provides the following key features (Sections 3 and 4):
   (a) A suite of benchmarks with different types of computations and input and output sizes.
   (b) Variable granularity response time measurement with operating system and Java language timers.
   (c) The integration of the HotSpot™ Java virtual machine monitoring and measurement infrastructure [3] so that response time can be explained in light of memory subsystem behavior.
   (d) The use of standard network packet capture tools to measure the consumption of network resources.
   (e) Support for recent versions of Java sockets, XML-RPC, and the GNU/Linux operating system.

2. Empirical results that reveal fundamental trade-offs in response time when remote communication primitives transfer data between client and server JVMs running on the same computational node (Section 5).

2. MOTIVATING PRINCIPLES

2.1 Communication Primitives

The Java programming language provides a wide variety of RCPs that could support the intra-node message exchange shown in Figure 1. A local-remote system can use communication primitives such as the Java-Message Passing Interface (Java-MPI) [11, 19], Java remote method invocation (RMI) [12, 25], tuple spaces [4, 36], or JXTA (i.e., "Juxtapose") [26, 31]. Yet, this paper focuses on the performance evaluation of sockets and XML-RPC because these primitives respectively represent (i) a low-level and a high-level remote communication mechanism and (ii) a well established standard and a recently proposed popular alternative.[1] While sockets typically support high performance message passing [2], they may require a programmer to understand several low-level implementation details. Alternatively, even though XML-RPC may compromise a system's performance, it furnishes a high-level communication paradigm that enables programming language independence and the rapid implementation of a program [2]. While the `java.net` package provides a refined implementation of sockets, XML can still be directly integrated into the Java programming language [13]. Furthermore, XML-RPC libraries are available for Java and systems such as OpenDHT use this protocol to facilitate client communication [28].

The `java.net.Socket` class provides an endpoint for communication between two JVMs. The `bind` operation attaches a socket to a local address on a computational node. The `accept` method blocks the socket server until a connection from a client occurs and the `connect` operation allows the client to actively seek out a server at a specified address. Sockets also furnish `send` and `receive` operations that enable the transmission of data. Finally, the `close` method releases the socket's address and ensures that it is not available for further data transmission. Since there is often a need for connection-oriented and reliable intra-node communication, this paper evaluates the performance of socket-based data transfer when TCP/IP governs the communication.

The XML-RPC primitive performs remote procedure calls that marshall and unmarshall Java objects that are encoded in XML. The Apache XML-RPC 2.0 implementation provides a `WebServer` class that is instantiated on the side of the server and associated with a uniform resource locator (URL). All of the server's `public` methods are then available to clients through the `WebServer` that handles the XML parsing and local-remote communication. The XML-RPC client is responsible for creating an instance of the `XmlRpcClient` class that automatically binds to the server's URL. Finally, the client invokes `execute` with a textual description of the desired server method and the Java objects that will serve as parameters. This paper measures communication performance when XML-RPC uses HTTP and TCP/IP, due to the prevalence of this configuration.

[1]Since the benchmarking framework supports the integration of other RCPs, Section 7 notes that we intend to incorporate and evaluate other communication primitives in future work.

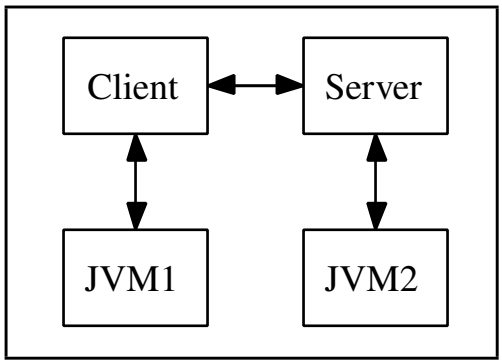


Figure 1: Intra-Node Communication.

2.2 Benchmarking Suites

Zhang and Seltzer observe that there are three main purposes for benchmarking: (i) comparing the performance of different systems, (ii) guiding performance optimizations, and (iii) predicting an application's performance in new software and hardware environments [37]. Furthermore, Zhang and Seltzer place software performance benchmarks into one of four categories: (i) micro, (ii) macro, (iii) combined, and (iv) application-specific benchmarks [37]. In the context of intra-node communication, this paper describes nano and micro benchmarks that measure the performance of remote communication primitives for Java.[2] Our nano benchmarks focus on the analysis of several basic operations that are commonly used in local-remote applications (e.g., transmission of a single value or a list of data values). The micro benchmarks incorporate small and well-defined operations (e.g., finding a data value in a list or reversing the provided list) that require server-side computations and thus may be found in a real-world local-remote system.

According to Horgan et al. [18], the performance evaluation of Java programs can take place at four different levels, namely (i) statically, with the source code; (ii) statically, with the bytecode; (iii) dynamically, with the bytecode; and (iv) dynamically, on a specific virtual machine and architecture. This paper focuses on the fourth category by analyzing the performance of Java-based RCPs on a specific JVM, computer architecture, and operating system kernel. The performance measurements that result from benchmark execution are further explained in light of JVM behavior profiles. Building on the insights developed by Heydon and Najork [17], this paper also examines how excessive heap allocation and the frequent execution of the garbage collector can impact the performance of intra-node communication.

3. BENCHMARKING FRAMEWORK

Figure 2 describes the *ExperimentCampaign* that executes N trials of benchmark B when it is configured to use communication primitive P. Line 1 of Figure 2 respectively initializes the tuples (i.e., ordered lists) of response times

[2]Since the metric prefix "nano" (10^{-9}) denotes a number that is smaller than the one described by the "micro" (10^{-6}) prefix, we use the term *nano benchmark* to refer to a benchmark that is smaller than its macro counterpart. We selected this descriptor instead of the previously used term *kernel benchmark* [1] since "kernel" now commonly refers to a part of an operating system.

Algorithm *ExperimentCampaign*(B, P, N)
Input: Benchmark B;
Remote Communication Primitive P;
Number of Experiment Trials N;
Output: Response Time Tuple $\mathcal{R}$;
Network Consumption Tuple $\mathcal{W}$;
JVM Profile Tuple $\mathcal{J}$
1. $\mathcal{R} \leftarrow \emptyset$; $\mathcal{W} \leftarrow \emptyset$; $\mathcal{J} \leftarrow \emptyset$;
2. *StartServer*(B, P)
3. *Pause()*
4. *StartPacketCapture*(B, P)
5. **for** $i \in \{1, \ldots, N\}$
6. **do** $\{R(B,P), J(B,P)\} \leftarrow$ *Execute*(B, P)
7. $\mathcal{R} \leftarrow \mathcal{R} \uplus \langle R(B,P) \rangle$
8. $\mathcal{J} \leftarrow \mathcal{J} \uplus \langle J(B,P) \rangle$
9. *Pause()*
10. *StopServer*(B, P)
11. $W(B,P) \leftarrow$*StopPacketCapture*(B, P)
12. $\mathcal{W} \leftarrow \langle W(B,P) \rangle$
13. **return** $\mathcal{R}, \mathcal{W}, \mathcal{J}$

Figure 2: The *ExperimentCampaign* Algorithm.

($\mathcal{R}$), network resource consumptions ($\mathcal{W}$), and JVM behavior profiles ($\mathcal{J}$) so that they can be populated with measurements during the execution of the experiments. The socket or XML-RPC server starts on line 2 and then the framework pauses so that the server can enter a quiescent state and the packet capture tool can initialize. Lines 5 through 9 ensure that (i) benchmark B executes for N trials, (ii) the response time and JVM profiles are properly stored for later analysis, and (iii) the server pauses in preparation for the next benchmark trial. These lines use the order preserving union operator, denoted $\uplus$, to add data values to $\mathcal{R}$ and $\mathcal{J}$. In an attempt to minimize the startup time of the framework, the server and the packet capture tool run continuously for the duration of the N trials (exploratory experiments indicated that all of the empirical results were not sensitive to this optimization). Finally, *ExperimentCampaign* terminates when line 13 returns the evaluation metrics.

Table 1 lists the four nano benchmarks that are similar to those that are described by Allman [2]. These nano benchmarks perform minimal computation on the side of the server in order to provide a baseline for the response time measurements. The `SS` benchmark uses a client that sends and receives a single data value, which is either an integer or a string. The client for `SV` sends a single data value and receives a vector of values. The `VS` benchmark has a client that sends a vector of values and receives a single value. Finally, the `VV` benchmark instructs the client to send and receive a vector of data values. During *ExperimentCampaign* the size of the vector of values, denoted $size(V)$, is fixed throughout every execution trial of `SV`, `VS`, and `VV`.

This paper also uses the four micro benchmarks listed in Table 2. After `GRAB`'s server randomly generates a vector of integers, the client transmits a single integer to the server. This benchmark is of type `SS` since we configured the `GRAB` server to use the client's integer as an index into its vector and subsequently return the resulting integer. `FACT` is an `SV` benchmark because it sends an integer value to a server that enumerates all the factors of the integer and returns a vector of integers. The type `VS` benchmark called `GCD` sends a vector of integers to a server that calculates the greatest common divisor and returns this integer to the client. The final benchmark is `REV`, which sends a vector of integers to the server. This benchmark is of type `VV` because the server reverses the order of the elements within the vector and returns this new vector back to the client.

Experiment	Sent by client	Received by client
`SS`	Single value	Single value
`SV`	Single value	Vector
`VS`	Vector	Single value
`VV`	Vector	Vector

Table 1: Nano Benchmarks.

Experiment	Sent by client	Received by client
`GRAB` (`SS`)	Single value	Single value
`FACT` (`SV`)	Single value	Vector
`GCD` (`VS`)	Vector	Single value
`REV` (`VV`)	Vector	Vector

Table 2: Micro Benchmarks.

4. EXPERIMENT GOALS AND DESIGN

4.1 Evaluation Metrics

The primary goal of the experiments is to measure the time overhead and network consumption associated with socket and XML-RPC communication. Equation (1) defines $R(B,P)$, the response time associated with the execution of a benchmark B from either Table 1 or 2 and a remote communication primitive P that is either sockets (S) or XML-RPC (X). For instance, $R(\texttt{GCD}, X)$ denotes the response time for the execution of the `GCD` benchmark with the XML-RPC primitive. Equation (2) describes $R_\Delta(B,P,P')$, the change in response time when we replace communication primitive P with primitive P'. Next, Equation (3) defines $R^{\%}_\Delta(B,P,P')$, the percent change in response time when the benchmark B uses P' instead of P. For example, $R^{\%}_\Delta(\texttt{FACT}, S, X)$ stands for the percent change in response time when `FACT` uses XML-RPC rather than sockets. Since it is often useful to calculate how response time changes when P' replaces P for a set of benchmarks designated β (e.g., all of the micro benchmarks in Table 2), Equation (4) characterizes $\bar{R}^{\%}_\Delta(\beta,P,P')$, the average percent change in response time over all of the benchmarks $B \in \beta$.

$$R(B,P) = T_{complete}(B,P) - T_{start}(B,P) \quad (1)$$

$$R_\Delta(B,P,P') = R(B,P') - R(B,P) \quad (2)$$

$$R^{\%}_\Delta(B,P,P') = \frac{R_\Delta(B,P,P')}{R(B,P)} \times 100 \quad (3)$$

$$\bar{R}^{\%}_\Delta(\beta,P,P') = \frac{\sum_{B \in \beta} R^{\%}_\Delta(B,P,P')}{|\beta|} \quad (4)$$

The benchmarking framework measures response time with timers that operate at two distinct levels of granularity. We leverage the operating system tool `/usr/bin/time` to record the coarse granularity time overheads. These operating system-based response times include the time required to start the client, communicate with the server, and shutdown the client. The benchmarks also use Java language instrumentation within the client's source code to measure

the fine granularity response times. The instrumentation records $T_{start}(B, P)$ before calling the server and then saves $T_{complete}(B, P)$ after the server finishes the computation.

Unless specified otherwise, this paper always reports the change and percent change in response time when the socket communication primitive is replaced with XML-RPC. For each of the benchmarks described in Section 3, the results analysis in Section 5 always reports $R_{\Delta}(B, S, X)$ and $R^{\%}_{\Delta}(B, S, X)$. A positive value for $R^{\%}_{\Delta}(B, P, P')$ indicates that response time increased when B used primitive P' instead of P. Alternatively, a negative value signals a decrease in response time when B uses P' rather than P. The framework also use the `jvmstat` and `hprof` tools to generate each JVM behavior profile in $\mathcal{J}$. Finally, the benchmarking framework employs Roubtsov's object sizing technique [29] to calculate the size of the method parameters and return values and, for each benchmark B and primitive P, it appends these values to the appropriate profile in $\mathcal{J}$.

Even though our focus is on intra-node communication, the use of HTTP and TCP/IP requires the client and server to transfer messages across the network interface. Therefore, we configured the `tcpdump` tool to receive and attempt to record all of the packets that are transmitted across the network interface and we use the `capinfos` tool to analyze the captured packets. The data reported by `tcpdump` and `capinfos` includes the (i) total number of packets received by the kernel filter, (ii) number of packets captured and stored for later analysis, and (iii) average packet size. Due to the high rate of data transfer, the number of captured packets can be much lower than the number of packets that were received by the `tcpdump` filter. Since `capinfos` can only analyze the captured packets, it is not always possible to accurately measure the total amount of network resources that the benchmark consumed during execution.

The benchmarking framework estimates the total network consumption in bytes, defined as $W(B, P, \Phi_c, \Phi_r)$ in Equation (5), by multiplying the average size of the captured packets by the total number of packets received by the `tcpdump` filter. Equation (5) uses Φ_c and Φ_r to respectively stand for the sets of captured and received packets. Moreover, this equation uses the *size* function to return the size in bytes for a given packet $\phi \in \Phi_c$. $W(B, P, \Phi_c, \Phi_r)$ is only an estimate of the actual network consumption since it assumes that all packets that were received, but not captured and analyzed, are of the average size of those packets $\phi \in \Phi_c$. While the use of $W(B, P, \Phi_c, \Phi_r)$ is a reasonable first step towards approximating network consumption, Section 7 suggests ways to avoid this estimation of network resource consumption and thus improve future empirical studies of Java's remote communication primitives.

$$W(B, P, \Phi_c, \Phi_r) = \frac{\sum_{\phi \in \Phi_c} size(\phi)}{|\Phi_c|} \times |\Phi_r| \quad (5)$$

4.2 Experiment Design and Analysis Methods

All experiments were conducted on a GNU/Linux workstation with kernel 2.6.12-1.1372, a dual-core 3 GHz Intel Pentium 4 processor, 1 GB of main memory, and 1 MB of L1 cache. The workstation used a Serial ATA connection to the hard drive and CPU hyper-threading was enabled in order to support thread-level parallelism. The experiments use a JVM version 1.5.0_02 which was set to operate in Java HotSpot^{TM}client mode with a 64 MB heap. Since our focus is on intra-node communication, we ran the benchmark client and server in separate JVMs on the same computational node. If desired, the benchmarking framework can be configured so that the client and server execute on separate nodes. We implemented XML-RPC communication with Apache XML-RPC 2.0 and the socket-based benchmarks use the classes provided by the `java.net` package. For the results in Section 5, we initialized *ExperimentCampaign* so that $N = 10$. We also set $size(V) = 5$ in order to produce the empirical results described in Section 5.1.

Section 5 analyzes the N response time measurements in $\mathcal{R}$ with descriptive statistics such as the arithmetic mean, denoted $R_{\mu}(\mathcal{R}, B, P)$ and defined in Equation (6). Figures 3 through 5 graphically depict the value of $R_{\mu}(\mathcal{R}, B, P)$ by the height of the corresponding bar. We also measure the dispersion of the N response times in the tuple $\mathcal{R}$ by calculating the standard deviation, designated as $R_{\sigma}(\mathcal{R}, B, P)$ and defined in Equation (7). The graphs in Figures 3 through 5 use an error bar to demarcate the range of values in the closed interval $[R_{\mu}(\mathcal{R}, B, P) - R_{\sigma}(\mathcal{R}, B, P), R_{\mu}(\mathcal{R}, B, P) + R_{\sigma}(\mathcal{R}, B, P]$. Finally, Figures 3 through 5 use a diamond at the top of a bar to signal that the value of $R_{\sigma}(\mathcal{R}, B, P)$ is too small to graphically present.

$$R_{\mu}(\mathcal{R}, B, P) = \frac{\sum_{R(B,P) \in \mathcal{R}} R(B, P)}{|\mathcal{R}|} \quad (6)$$

$$R_{\sigma}(\mathcal{R}, B, P) = \sqrt{\frac{\sum_{R(B,P) \in \mathcal{R}} (R(B, P) - R_{\mu}(\mathcal{R}, B, P))^2}{|\mathcal{R}|}} \quad (7)$$

Whenever the remote communication primitives demonstrate performance characteristics of the greatest similarity, we also calculate a mean confidence interval and perform a mean difference hypothesis test. Within a group of similar benchmarks β, we perform additional statistical analysis for a benchmark $B \in \beta$ and the two primitives P and P' whenever (i) $R(B, P') - R(B, P)$ is the smallest or (ii) either $R(B, P')$ or $R(B, P)$ has the largest standard deviation. For instance, the results in Figure 3(a) prompted us to compare (i) S-`SS` to X-`SS` because $.277 - .17 = .107$ is the smallest difference between two benchmarks and (ii) S-`VV` to X-`VV` since X-`VV` shows the largest standard deviation. Table 5 summarizes the confidence intervals that we calculated.

We apply these additional statistical analyses with the observations that (i) the response times adhere to an interval measurement scale, (ii) the N samples within $\mathcal{R}$ are independent, and (iii) the variances of the response times for the two different communication primitives are not equal. Observation (i) is justified because the difference between two response time measurements is meaningful. That is, if response time is measured in seconds and we have $R_1(B, P) = .15$ and $R_2(B, P) = .10$, then we can accurately say that R_1 is .05 seconds slower than R_2. We judge that observation (ii) is valid because *ExperimentCampaign* includes configurable pauses between each execution trial of a benchmark (we set the pause to five seconds for all of the empirical results in Section 5, although this time period is a configurable parameter of the framework). Since the response time results also demonstrate that the variances were not always exactly equal, as stated in observation (iii), we employ the Welch's approximate t-test rather than the traditional t-test.

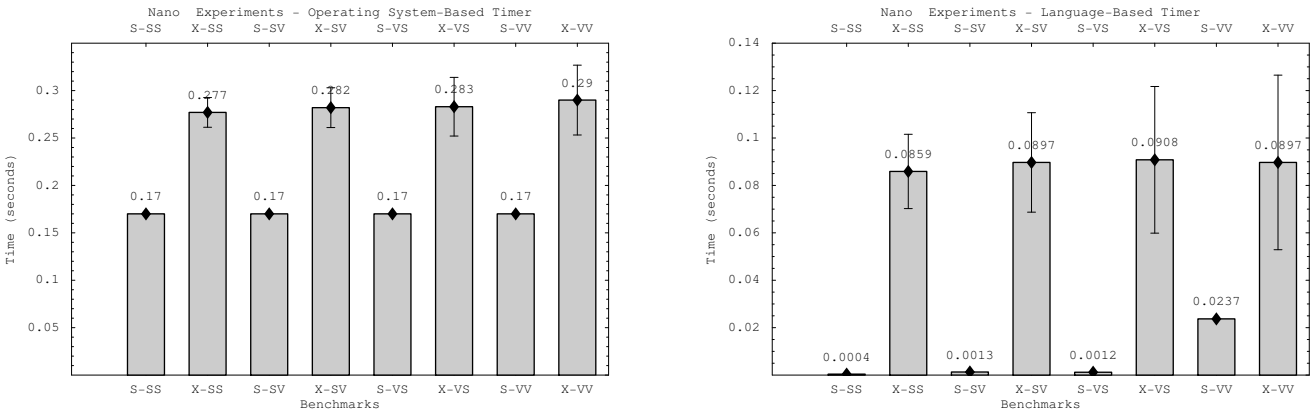


Figure 3: Nano Benchmarks Using the (a) Operating System-Based and (b) Language-Based Timers.

For a benchmark B and two remote communication primitives P and P', we formulate the null hypothesis as H_0 : $R_\mu(B,P) = R_\mu(B,P')$. A rejection of H_0 suggests that communication primitives P and P' have different response time characteristics and, all other factors being equal, we would prefer the primitive with the lower response time values. Since we configured the t-test with a significance level of .01 (i.e., $\alpha = .01$), the data in Table 5 represents the 99% confidence interval (CI) for the arithmetic mean of the N response times. For each response time mean in the confidence interval $[l,u]$, we can be 99% certain that the mean of the response time values from subsequent experiments will fall between the lower bound l and the upper bound u. Therefore, small confidence intervals suggest that our benchmarking framework generates empirical outcomes in a repeatable and predictable manner.

5. EXPERIMENTAL RESULTS

5.1 Response Time

Nano Benchmarks. Figure 3(a) presents the results from measuring response time with the operating system-based timer when $size(V) = 5$. The most important trend to note is that socket communication is always faster than XML-RPC. For instance, the average across all benchmarks for sockets was 0.17 seconds compared to 0.283 seconds for XML-RPC. It is further evident that $R^{\%}_{\Delta}(\texttt{VV}, S, X)$= 70.58% and there is no overlap in the standard deviation for each benchmark. As shown in Table 5, the socket implementation of `SS` had a confidence interval of [0.17,0.17], while the interval for XML-RPC's response times was [0.269,0.293]. These results are significant because the confidence intervals do not overlap and the result of the t-test was to reject the null hypothesis. Lastly, the summary information in Table 5 also demonstrates that sockets and XML-RPC have statistically different response times for the `VV` benchmark.

Figure 3(b) provides the measurements obtained with the language-based timer. These results suggest that sockets are both faster and more predictable than XML-RPC. For example, the average across all benchmarks was 0.00665 seconds for sockets and 0.089025 seconds for XML-RPC. We also observe that $R^{\%}_{\Delta}(\texttt{VV}, S, X)$= 278.48% and there is no overlap in the standard deviation for each benchmark's time overhead. Finally, Table 5 reveals that the communication primitives have different performance characteristics

Descriptor	$size(V)$ (count)	$size(V)$ (bytes)
Small	5	140
Medium	50	960
Large	500	8600

Table 3: *size(V)* Variation.

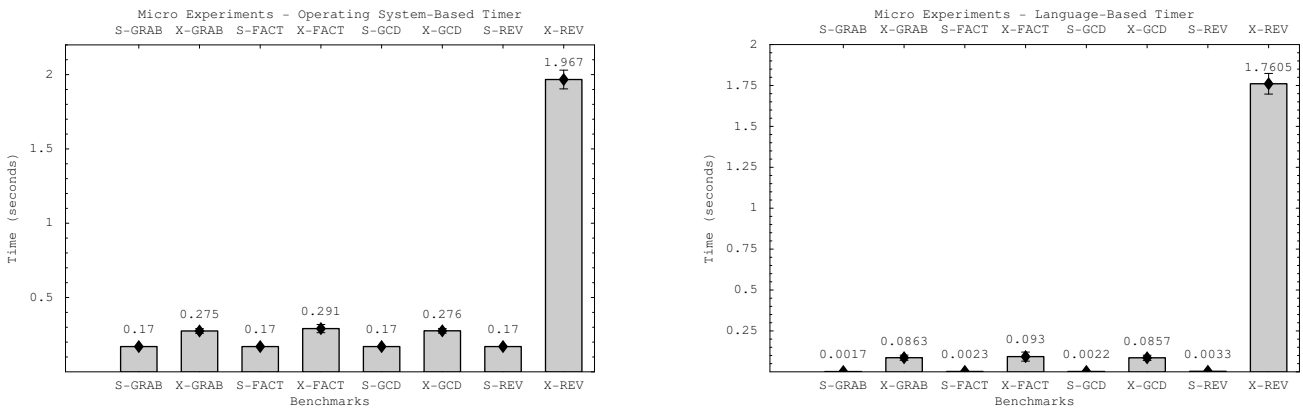
since (i) the socket implementation of `VV` has a confidence interval of [0.023,0.025], (ii) XML-RPC has an interval of [0.072,0.107], and (iii) the t-test rejects the null hypothesis.

Micro Benchmarks. Figure 4(a) gives the results from measuring response time with the operating system-based timer when $size(V) = 5$. Across all micro benchmarks, we note that the mean time overhead for sockets was 0.17 seconds, the XML-RPC primitive yielded an average response time of 0.702 seconds, and we further calculate that $\bar{R}^{\%}_{\Delta}(B,S,X)$= 312.94%. For the `REV` benchmark, Table 5 shows that the t-test rejects H_0 and that sockets and XML-RPC have confidence intervals of [0.17,0.17] and [1.902,2.039], respectively. While Table 5 indicates that the confidence intervals for `GRAB`'s response times do overlap, the t-test still rejects the null hypothesis and thus we judge that the primitives have different time overheads. In this case, the t-test rejects H_0 because the socket implementation of `GRAB` exhibits no variability in the response time metric.

Figure 4(b) depicts the results that were obtained from measuring response time with the language-based timer. The average across all benchmarks for sockets was 0.002375 seconds and 0.506375 seconds for XML-RPC and as such $R^{\%}_{\Delta}(B,S,X)$= 213.11%. Table 5 confirms the same empirical trend: for the `GRAB`, `GCD`, and `REV` benchmarks and the language timers, sockets are faster than XML-RPC in a statistically significant manner. In summary, the nano and micro benchmarks indicate that the transmission of small vector and integer parameters (i.e., $size(V) = 5$) causes the socket communication primitive to exhibit response times that are lower and less variable than XML-RPC's.

5.2 Vector Size Variation

Since real-world Java applications that perform intra-node communication may transmit vectors of different sizes, we conducted additional experiments to determine how the variation of vector size impacted response time. Table 3 gives the different values of $size(V)$ for each execution of *ExperimentCampaign*. Figure 5(a) furnishes the results from varying $size(V)$ with `SV`. The experiments indicate that



(a) (b)

Figure 4: Micro Benchmarks Using the (a) Operating System-Based and (b) Language-Based Timers.

$R^{\%}_{\Delta}(\texttt{SV}, S_5, S_{500})$= 4069.00% and $R^{\%}_{\Delta}(\texttt{SV}, X_5, X_{500})$= 53.39%.[3] The very low response time associated with socket-based communication when $size(V) = 5$ causes the large percent increase when S_{500} is executed instead of S_5. Yet, calculations with and study of the results in Figure 5(a) reveal that $R^{\%}_{\Delta}(\texttt{SV}, S_{500}, X_{500})$= 141.60% and thus sockets still exhibit lower and less dispersed response times than XML-RPC. The summary results in Table 5 confirm this trend since sockets and XML-RPC have respective confidence intervals of [0.050,0.058] and [0.106,0.156] and the t-test rejects H_0. Overall, Figures 5(a) through 5(c) and Table 5 support the conclusion that sockets are faster than XML-RPC when the maximum value of $size(V)$ is less than or equal to 500.

Table 4 gives the results from experiments that increased $size(V)$ to very large values during the execution of `VV`. Overall, sockets are faster than XML-RPC when $size(V) =$ 5000, whereas XML-RPC demonstrates a slight performance advantage when the vectors have 10000 items. Further analysis reveals that $R^{\%}_{\Delta}(\texttt{VV}, S_{5000}, X_{5000})$ =16.44% and the two primitives have overlapping confidence intervals of [0.292,0.304] for sockets and [0.292,0.401] for XML-RPC. Even though these intervals overlap, the result of the t-test was to reject the null hypothesis and we judge that the socket primitive has better performance. Figure 6 provides an empirical cumulative distribution function (ECDF) that explains why the two communication primitives have different response time characteristics when `VV` is executed with $size(V) = 5000$. The ECDF curve represents the probability that the response time is less than or equal to a specific value on the horizontal axis. Figure 6 shows that $R(\texttt{VV}, S_{5000})$ is always less than .309 seconds while only 80% of the $R(\texttt{VV}, X_{5000})$ values fall between .316 and .35 seconds.

Executing `VV` with $size(V) = 10000$ exposes the fact that $R^{\%}_{\Delta}(\texttt{VV}, S_{10000}, X_{10000})$ =−12.54% and thus XML-RPC exhibits lower response times than sockets. The confidence intervals for sockets and XML-RPC were respectively [0.588,0.608] and [0.469,0.577] and the result of the t-test was to reject the null hypothesis. Table 4 also shows that the results for $size(V) = 50000$ were more pronounced and thus $R^{\%}_{\Delta}(\texttt{VV}, S_{50000}, X_{50000})$ =−90.96%. The confidence interval for the responses times of the socket implementation was

[3]In the remainder of this paper the notation P_k denotes the use of remote communication primitive P to transmit a vector with k entries. For example, S_{500} means that the socket primitive transmitted a vector V of 500 values (i.e., $size(V) = 500$).

$size(V)$	$size(V)$ (bytes)	$R(\texttt{VV}, S)$ (sec)	$R(\texttt{VV}, X)$ (sec)
5000	80,520	0.298	0.347
10000	161,000	0.598	0.523
50000	927,720	18.784	1.697

All Response Times Calculated by the Language-Based Timers

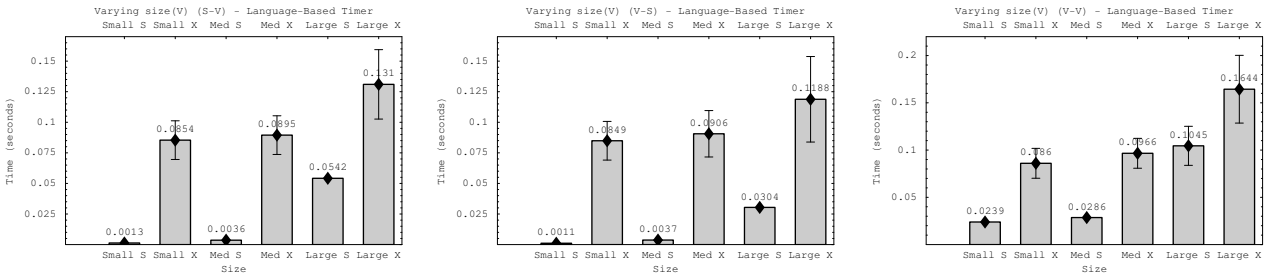
Table 4: Response Time with Very Large Vectors.

[18.078,19.490], the interval for XML-RPC was [1.616,1.777], and the t-test rejected the null hypothesis. These results clearly demonstrate that sockets perform worse than XML-RPC when transmitting large parameters and return values.

We explored a wide range of parameter settings in order to ensure that this marked performance change was not due to an improper configuration of sockets. For instance, increasing the size of the socket server's JVM heap to 256 MB or modifying the size of the socket's send and receive buffers with the `setSendBufferSize` and `setReceiveBufferSize` methods provided by `java.net.Socket` did not change the results in Table 4. We also used the `setPerformancePreferences(CON, LAT, BAN)` to modify the performance preferences for the socket primitive and this did not change the results for the socket-based version of `VV`. The parameters `CON`, `LAT`, and `BAN` respectively indicate the relative importance of short connection time, low latency, and high bandwidth. Yet, response times did not differ when we called `setPerformancePreferences(0,1,0)` in order to favor low latency over connection time and bandwidth. This exploration strengthens the conclusion that sockets are truly slower than XML-RPC when transferring bulk data.

5.3 Virtual Machine Behavior

This paper uses JVM behavior profiles to explain the response time characteristics revealed in Sections 5.1 and 5.2. In particular, we focus on a benchmark's heap allocation behavior and the execution of the JVM's garbage collector. The JVM heap is divided into several regions respectively known as permanent, old, and young. The young space is further sub-divided into three regions known as eden, survivor one (S1), and survivor two (S2). We record the number of times the collector performs a full collection over all three regions and a young collection for eden, S1, and S2. While young GC (YGC) events often occur more frequently than the full GC (FGC) events, a full GC normally takes longer.



(a) (b) (c)

Figure 5: Varying *size(V)* for the (a) SV, (b) VS, and (c) VV Nano Benchmarks.

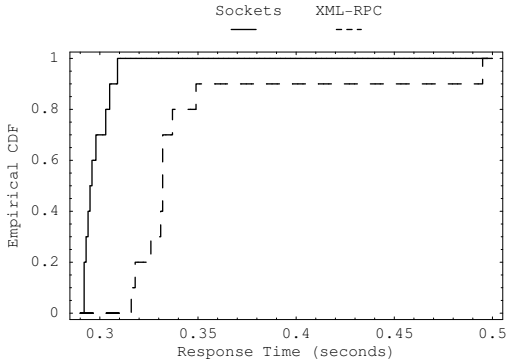


Figure 6: Response Time for VV when $size(V) = 5000$.

An intuitive explanation of how we implemented the socket and XML-RPC benchmarks aids in understanding the meaning of the behavior profiles. During the transfer of parameters and return values, a socket benchmark uses the `writeChars` method of the `java.io.DataOutputStream` and the `readLine` operation from `java.io.DataInputStream`. While a socket client calls `writeChars` to send data to a server and `readLine` to receive the server's return value, the server uses `readLine` to get the parameters and `writeChars` to transfer the return value. Before transmission, an XML-RPC client encodes the Java object parameters in the character-based XML format, whereas the server must decode the XML message in order to transform the text back into objects. The XML-RPC server and client reverse this procedure to respectively handle the encoding and decoding of the return value. Overall, both of the benchmarks process characters and strings since they use a textual representation for the parameters and return values. Yet, the frequent conversion between Java objects and text and the use of strings and characters can often cause the benchmarks to inefficiently allocate many data values to the heap [21].

Figure 4 shows that the `REV` benchmark takes longer to complete when it uses XML-RPC instead of the socket primitive. This is due in part to the fact that the socket primitive does not cause any YGC or FGC events to occur. The XML-RPC primitive forces 122 YGC events that consume a total of .059 seconds and 5 FGC events that increase response time by .123 seconds. Yet, the XML-RPC server must also perform additional string processing in order to reverse the input vector and this increases the response time as well. Table 4 shows that the execution of `VV` can yield very high response times for sockets and Table 6 reveals that sockets and XML-RPC have similar GC behavior when $size(V) = 5000$ and $size(V) = 10000$. For example, S-`VV` and X-`VV` both avoid a full garbage collection when 5000 objects are transmitted. Even though sockets do require more GC activity when $size(V) = 10000$, the YGC and FGC events only increase response time by .073 seconds.

Tables 6(a) and (b) indicate that S-`VV`'s JVM performs 1645 YGC and 663 FGC events when $size(V) = 50000$, while X-`VV` only causes 123 YGC and 5 FGC events. The 10.375 seconds associated with performing all of the full GC events represent 55.23% of the execution time for S-`VV`. In contrast, the FGC events performed by the X-`VV` JVM correspond to 8.43% of the benchmark's total execution time. The use of the JVM heap profiling agent, called `hprof`, reports that the S-`VV` JVM allocates 152,539 objects to the heap while 1,506,476 objects are allocated by the X-`VV` JVM. However, the S-`VV` benchmark allocates 710,374,184 bytes and the X-`VV` JVM only stores 54,101,312 bytes. At the point of benchmark termination, S-`VV` has 4,773,224 bytes of live objects in the JVM heap and the X-`VV` heap contains 7,234,520 bytes of live objects. Finally, the JVM behavior profiles reveal that the `java.net.Socket` implementation allocates many `char[]` arrays while the XML-RPC primitive relies upon instances of the high-performance `java.nio.CharBuffer`. These results further support the conclusion that S-`VV` is slow because (i) it allocates and subsequently collects many more bytes than X-`VV` and (ii) sockets in Java 1.5 do not take advantage of the fast character buffers in Java's "new input/output" (NIO) package.

5.4 Network Resource Consumption

Table 7 shows the network resource consumptions for the nano benchmarks with $size(V) = 500$. We do not report consumption measurements for benchmarks that were executed with large $size(V)$ values since very high data transfer rates caused the `tcpdump` kernel filter to drop a significant number of packets and this could yield incorrect estimates for W. The most obvious trend to note is that socket communication always generates more packets than XML-RPC. Across all benchmarks, the average number of packets received by the GNU/Linux kernel was 7222 for the socket primitive and 380 for XML-RPC. We also observe that the average packet size is smaller for sockets than XML-RPC. Considering every benchmark, the mean of the average packet sizes is 86 bytes for the socket primitive and 802 bytes for the XML-RPC communication mechanism.

Using the approach and equations in Section 4.1, we calculated that $W^{\%}_{\Delta}(\texttt{SS}, S, X)$ =30.90%, $W^{\%}_{\Delta}(\texttt{SV}, S, X)$ =118.86%, $W^{\%}_{\Delta}(\texttt{VS}, S, X)$ =−68.73%, and $W^{\%}_{\Delta}(\texttt{VV}, S, X)$ =−44.71%. These percent increases suggest that socket communication

Comparison	CI for Sockets	CI for XML-RPC	Instrumentation	
			Language	System
$R(\texttt{SS}, S_5)$ vs $R(\texttt{SS}, X_5)$	[0.17, 0.17]	[0.269, 0.293]		✓
$R(\texttt{VV}, S_5)$ vs $R(\texttt{VV}, X_5)$	[0.17, 0.17]	[0.252, 0.328]		✓
$R(\texttt{VV}, S_5)$ vs $R(\texttt{VV}, X_5)$	[0.023, 0.025]	[0.072, 0.107]	✓	
$R(\texttt{REV}, S_5)$ vs $R(\texttt{REV}, X_5)$	[0.17, 0.17]	[1.902, 2.039]		✓
$R(\texttt{GRAB}, S_5)$ vs $R(\texttt{GRAB}, X_5)$	[0.17, 0.17]	[0.159, 0.191]		✓
$R(\texttt{GCD}, S_5)$ vs $R(\texttt{GCD}, X_5)$	[0.002, 0.003]	[0.067, 0.104]	✓	
$R(\texttt{REV}, S_5)$ vs $R(\texttt{REV}, X_5)$	[0.002, 0.004]	[1.687, 1.834]	✓	
$R(\texttt{SV}, S_{500})$ vs $R(\texttt{SV}, X_{500})$	[0.050, 0.058]	[0.106, 0.156]	✓	
$R(\texttt{VS}, S_{500})$ vs $R(\texttt{VS}, X_{500})$	[0.019, 0.042]	[0.089, 0.148]	✓	
$R(\texttt{VV}, S_{500})$ vs $R(\texttt{VV}, X_{500})$	[0.090, 0.119]	[0.128, 0.201]	✓	
$R(\texttt{VV}, S_{5000})$ vs $R(\texttt{VV}, X_{5000})$	[0.292, 0.304]	[0.292, 0.401]	✓	
$R(\texttt{VV}, S_{10000})$ vs $R(\texttt{VV}, X_{10000})$	[0.588, 0.608]	[0.469, 0.577]	✓	
$R(\texttt{VV}, S_{50000})$ vs $R(\texttt{VV}, X_{50000})$	[18.078, 19.490]	[1.616, 1.777]	✓	

For Each Row, the t-test Rejected H_0 at a Significance Level of 0.01

Table 5: Summary Table for the Confidence Intervals and Hypothesis Tests.

consumes fewer network resources than XML-RPC for the `SS` and `SV` benchmarks. As given in Table 7, the results with `VS` and `VV` reveal that socket communication transmits more data than XML-RPC. For instance, we estimate that S-`VS` transmits a total of $13944 \times 71.16 = 992,255.04$ bytes whereas X-`VS` only transfers $360 \times 861.67 = 310,201.20$ bytes during execution. This outcome suggests that, at the packet level, XML-RPC's textual encoding of vectors can be more space efficient than the Java serialization mechanism used by sockets. However, the XML-RPC primitive might not be suitable for interactive Java applications because it frequently causes the transmission of very large packets (e.g., X-`VV` has an average packet size of 1455.90 bytes).

5.5 Threats to Validity

Any empirical study of the performance of local-remote software systems must confront certain threats to validity. During the empirical evaluation of communication primitive performance we were aware of potential threats to validity and we took steps to control the impact of these threats. Threats to internal validity concern factors that would present alternative explanations for the empirical results discussed in Section 5. The first threat to internal validity is related to the potential faults within the benchmarking framework that Section 3 describes. We controlled this threat by incorporating tools and libraries that are frequently used by practitioners, such as `tcpdump`, `capinfos`, `jvmstat`, `hprof`, and the `java.net` package. Since we have repeatedly used these tools without experiencing errors or anomalous results, we have a confidence in their correctness and we judge that they did not negatively impact the validity of our empirical study. Moreover, Allman also leveraged tools such as `tcpdump` without noticing problems that would compromise his results [2]. We also tested each benchmark in isolation in order to ensure that it regularly produced meaningful outcomes. Finally, we controlled threats to internal validity by using the same workstation and preventing additional user logins throughout experimentation.

Threats to external validity would limit the generalization of our approach and the empirical results to new benchmarks and execution environments. The experiments in this paper only focus on the use of four nano and four micro benchmarks and the performance measurements from these studies might be different from those produced by other nano, micro, macro, combined, and application-specific benchmarks. The experiments only evaluate sockets and XML-RPC and thus they provide no direct insights into the behavior of local-remote systems constructed with other communication primitives. Also, the client and server in the current benchmarks exchange a limited variety of parameters and return values. However, our framework supports the integration of other benchmarks and this paper reports on the results from using benchmarks that are similar to those that were described by Allman [2]. Another threat to external validity is related to the fact that the experiments measure the performance of sockets and XML-RPC in a single execution environment. This is because the computer hardware, JVM, operating system kernel, and other environmental factors were not varied during experimentation. Yet, it is our judgment that the execution environment is representative of one that is frequently used during the development and execution of a local-remote system.

Threats to construct validity concern whether the evaluation metrics accurately reflect the variables that the experiments were designed to measure. We judge that the response time metric is defined and measured in a fashion that will be useful to individuals who implement applications that perform intra-node communication. While network resource consumption is also a valuable metric, our current tools do not always support the accurate measurement of $W(B, P)$. As noted in Section 4.1, this is due to the fact that the high rates of data transfer rapidly exhaust the kernel buffer space that is reserved for packet capture. Before conducting further experiments we will attempt to control this threat to validity by modifying the GNU/Linux kernel and/or enhancing existing packet capture tools.

6. RELATED WORK

The focus of this paper connects to prior work in the areas of the implementation of remote communication primitives and the benchmarking of sockets and XML-RPC.

Communication Primitives. The micro-kernel is an example of an operating system whose architecture was driven by detailed empirical evaluations of communication

$size(V)$	YGC Events (count)	YGC Time (sec)	FGC Events (count)	FGC Time (sec)
5000	16	.008	0	0
10000	63	.023	4	.050
50000	1645	.697	663	10.375

(a)

$size(V)$	YGC Events (count)	YGC Time (sec)	FGC Events (count)	FGC Time (sec)
5000	14	.016	0	0
10000	27	.022	1	.020
50000	123	.695	5	.143

(b)

Table 6: VV's Garbage Collection Behavior for (a) Sockets and (b) XML-RPC.

primitive performance [10, 22, 23]. Since interprocess communication (IPC) was often the limiting factor in microkernel performance, developers designed numerous benchmarks and used response time results to guide the design of IPC protocols [14, 22]. These studies are similar to the one in this paper since we also use empirical results to motivate the choice between primitives for intra-node communication. Furthermore, Bershad et al. propose an interesting and efficient mechanism to support intra-node communication with lightweight RPCs [5]. While our focus is on modern operating systems, object-oriented languages, and virtual machines, the LRPC primitive was implemented for both the Taos operating system and the DEC Firefly multiprocessor [5] and a previous version of the Mach kernel [7]. Finally, both Herder et al. and Karger have developed specialized implementations and optimizations that improve the performance of intra-node communication [15, 20]. In contrast to these efforts, this paper considers the evaluation of primitives that are already in wide use and reputed to be easy to program, namely Java sockets and XML-RPC.

Sockets and XML-RPC. This paper is directly related to Allman's empirical analysis of the socket and XML-RPC primitives [2]. Yet, this article is different than [2] because it (i) reports benchmark results with more recent versions of sockets and XML-RPC, (ii) solely focuses on experiments where the client and server reside on the same node, (iii) measures response time with OS and language-based timers, and (iv) explains the results in light of JVM behavior. It is interesting to note that the results of this paper contrast with those reported by Allman. The previous work suggests that sockets and XML-RPC have similar response time characteristics for small transactions. However, the past experiments with larger transactions show that sockets perform up to an order of magnitude better than XML-RPC [2]. Our experiments demonstrate that sockets outperformed XML-RPC with smaller transactions but that at extremely large transaction sizes XML-RPC performed significantly better than sockets. We judge that the conflicting performance characterizations could be attributed to the (i) different execution environments (e.g., kernel, OS, and JVM), (ii) different versions of sockets and XML-RPC, and (iii) location of the client and server JVMs. The future experimentation described in Section 7 will support the resolution of the differences between these two studies.

7. CONCLUSIONS AND FUTURE WORK

This paper describes a benchmarking framework that supports the empirical evaluation of intra-node communication with Java-based primitives. The presented method yields quantitative results that characterize how and why sockets and XML-RPC consume time overhead and network resources. The use of the benchmarks in a specific execution environment yielded interesting experimental results that characterize the strengths and weaknesses associated with sockets and XML-RPC. Section 5.1 reveals that, for all of the benchmarks, socket-based communication outperforms XML-RPC when $size(v) = 5$. Yet, the results in Section 5.2 point out that when the nano benchmarks perform bulk data transfer, the performance of sockets degrades rapidly.

Benchmark	Packets Received	Packets Captured	Avg Packet Size (bytes)
S-SS	402	201	74.06
X-SS	360	180	108.28
S-SV	1180	518	121.14
X-SV	400	200	782.15
S-VS	13944	4008	71.16
X-VS	360	180	861.67
S-VV	13362	4044	78.83
X-VV	400	200	1455.90

All Benchmarks Use a Vector with 500 Values

Table 7: Network Resource Consumption.

Section 5.3 further interprets the response time results in light of the behavior of the JVM's garbage collector. An analysis of the JVM behavior profiles shows that sockets incur additional time overhead because they trigger a significant number of YGC and FGC events. Furthermore, the results in Section 5.4 suggest that (i) sockets generate a larger number of packets than XML-RPC and (ii) the average packet size created by the socket-based benchmarks is smaller than the packet size produced by XML-RPC. In conclusion, the results in this paper can serve as valuable guides when selecting Java-based primitives for intra-node communication. It is possible to extend our framework with new communication primitives, benchmarks, and statistical analysis techniques. Thus, future research, enabled by our method of approach, can continue to develop a deeper understanding of both general-purpose primitives in languages like Java (e.g., sockets and XML-RPC) and other customized intra-node communication schemes (e.g., [15, 16, 32, 33]).

In future research, we will modify and/or re-configure the GNU/Linux kernel and the packet capture tools, for instance by adding larger buffers to store additional packets, in or-

der to more accurately characterize network resource consumption. We also want to improve the presented work by introducing multi-threaded clients and measuring benchmark throughput. We will further extend the benchmarks so that they can transmit more realistic parameters and return values like complex Java objects and binary files. It would also be useful to run both the current and additional benchmarks on different architectures and platforms in order to determine how the execution environment impacts the performance results. Moreover, we will measure message passing response time when the client and server exist on separate machines. Since this communication will occur between non-local JVMs, we will study the performance of the primitives on congested and/or unreliable networks.

We intend to execute long running benchmark campaigns that invoke a series of methods on the side of the server while continuing to collect JVM behavior profiles. This will enable us to identify the impact that HotSpot™adaptive optimization has on the performance of the communication mechanisms. Additional research will also investigate alternatives to socket and XML-RPC communication such as Java-MPI [11, 19], Java RMI [12, 25], tuple spaces [4, 9, 36], and JXTA [26, 31]. These alternatives will be compared to sockets and XML-RPC in order to determine how each communication method compares to the others in terms of response time, network resource consumption, and JVM behavior. Ultimately, we want to extend our framework to also include the customized communication primitives used in other local-remote systems (e.g., [15, 16, 32, 33]).

8. REFERENCES

[1] R. C. Agarwal, B. Alpern, L. Carter, F. G. Gustavson, D. J. Klepacki, R. Lawrence, and M. Zubair. High-performance parallel implementations of the NAS kernel benchmarks on the IBM SP2. *IBM Sys. Jour.*, 34(2):263–272, 1995.

[2] M. Allman. An evaluation of XML-RPC. *Perf. Eval. Rev.*, 30(4), 2003.

[3] E. Armstrong. HotSpot: A new breed of virtual machine. *JavaWorld*, 1998.

[4] G. C. Arnold, G. M. Kapfhammer, and R. S. Roos. Implementation and analysis of a JavaSpace supported by a relational database. In *Proc. of 8th PDPTA*, 2002.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proc. of 12th SOSP*, 1989.

[6] W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *Proc. of 5th PPPJ*, 2007.

[7] V. Bourassa and J. Zahorjan. Implementing lightweight remote procedure calls in the Mach 3 operating system. Technical Report TR-95-02-01, 1995.

[8] P. Dasgupta, R. J. LeBlanc, Jr., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Comp.*, 24(11), 1991.

[9] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis. Towards the measurement of tuple space performance. *Perf. Eval. Rev.*, 33(3), December 2005.

[10] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *Proc. of 9th Euro. Work.*, 2000.

[11] V. Getov, P. Gray, and V. Sunderam. MPI and Java-MPI: contrasts and comparisons of low-level communication performance. In *Proc. of ICS*, 1999.

[12] W. Grosso. *Java RMI*. O'Reilly and Associates, Inc., Sebastopol, CA, USA, 2002.

[13] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In *Proc. of 14th WWW*, 2005.

[14] H. Hartig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schonberg. The performance of μ-kernel-based systems. *Oper. Sys. Rev.*, 31(5), 1997.

[15] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *Oper. Sys. Rev.*, 40(3), 2006.

[16] J. N. Herder, H. Bros, and A. S. Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Vrije Universiteit, 2006.

[17] A. Heydon and M. Najork. Performance limitations of the Java core libraries. In *Proc. of Java Grande*, 1999.

[18] J. Horgan, J. Power, and J. Waldron. Measurement and analysis of runtime profiling data for Java programs. In *Proc. of SCAM*, November, 2001.

[19] G. Judd, M. Clement, Q. Snell, and V. Getov. Design issues for efficient implementation of MPI in Java. In *Proc. of Java Grande*, 1999.

[20] P. A. Karger. Using registers to optimize cross-domain call performance. *Comp. Arch. News*, 17(2), 1989.

[21] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in Java strings. In *Proc. of 23rd OOPSLA*, 2008.

[22] J. Liedtke. Improving IPC by kernel design. In *Proc. of 14th SOSP*, 1993.

[23] J. Liedtke. Toward real microkernels. *Commun. of the ACM*, 39(9), 1996.

[24] G. Lowney. Why Intel is designing multi-core processors. In *Proc. of 18th SPAA*, 2006.

[25] J. Maassen, R. van Nieuwpoort, R. D. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *Trans. on Progr. Lang. and Sys.*, 23(6), 2001.

[26] S. Oaks and L. Gong. *JXTA in a Nutshell*. O'Reilly and Associates, Inc., Sebastopol, CA, USA, 2002.

[27] S. R. Radia, G. Hamilton, P. B. Kessler, and M. L. Powell. The Spring object model. In *Proc. of COOTS*, 1995.

[28] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: a public DHT service and its uses. In *Proc. of SIGCOMM*, 2005.

[29] V. Roubtsov. Sizeof for Java: Object sizing revisited. *JavaWorld*, 2003.

[30] M. Schlansker, N. Chitlur, E. Oertli, J. Paul M. Stillwell, L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi. High-performance Ethernet-based communications for future multi-core processors. In *Proc. of ICS*, 2007.

[31] J.-M. Seigneur, G. Biegel, and C. D. Jensen. P2P with JXTA-Java pipes. In *Proc. of 2nd PPPJ*, 2003.

[32] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Integrating remote invocation and distributed shared state. In *Proc. of IPDPS*, 2004.

[33] K. Vaidyanathan, P. Lai, S. Narravula, and D. K. Panda. Optimized distributed data sharing substrate in multi-core commodity clusters: A comprehensive study with applications. In *Proc. of CCGrid*, 2008.

[34] J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In *Proc. of Mob. Obj. Sys.*, 1997.

[35] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. of 5th CGO*, 2007.

[36] G. C. Wells. New and improved: Linda in Java. *Scien. of Comp. Progr.*, 59(1–2), January 2006.

[37] X. Zhang and M. Seltzer. HBench:Java: an application-specific benchmarking framework for Java virtual machines. In *Proc. of Java Grande*, 2000.