Requirements for and Evaluation of RMI Protocols for Scientific Computing*

Madhusudhan Govindaraju, Aleksander Slominski,
Venkatesh Choppella, Randall Bramley, Dennis Gannon
Department of Computer Science
Indiana University
Bloomington, IN

Abstract

Distributed software component architectures provide a promising approach to the problem of building large scale, scientific Grid applications [18]. Communication in these component architectures is based on *Remote Method Invocation* (RMI) protocols that allow one software component to invoke the functionality of another. Examples include Java remote method invocation (Java RMI)[25] and the new *Simple Object Access Protocol* (SOAP) [15]. SOAP has the advantage that many programming languages and component frameworks can support it. This paper describes experiments showing that SOAP by itself is not efficient enough for large scale scientific applications. However, when it is embedded in a multi-protocol RMI framework, SOAP can be effectively used as a universal control protocol, that can be swapped out by faster, more special purpose protocols when large data transfer speeds are needed.

Key Words: Distributed computing, software component systems, communication protocols, RMI, Java, SOAP.

1 Introduction

Distributed component systems [1, 23, 26, 27] for scientific and engineering computing can potentially provide the same benefits that components do for business and financial computing: seamless access to remote resources, plug-and-play software composition without recompilation, access to specialized non-compute resources like data warehousing and visualization, and improved software reuse. Component systems also can provide a natural milieu for large multidisciplinary research teams with pools of expertise distributed across the country. However, scientific computing presents challenges not typically found in commercial applications. In particular, the components encapsulate parallel programs sending large, complex, and rapidly changing data objects.

*0-7803-9802-5/2000/$10.00 ©2000 IEEE.

Because of this, the underlying communications substrate plays a more critical role than it does for commercial computing. The communication system must be capable of high-performance messaging, efficiently using specialized high-speed networks like Abilene, MREN, or ESNET when possible, using modern research protocols like quality of service and source routing, and handling parallel-to-parallel component communications. Modern communication systems are based on remote method invocation (RMI) protocols that allow an object in one address space to invoke the methods of an object in another address space. The desiderata for an effective RMI system include reliability, robustness, human accessibility, readily usable APIs in modern computer languages, interoperability across languages, platforms, and component frameworks, and integration with the emerging Grid infrastructure [18] and standards like COM, OMG's Corba Component Model and the DOE Common Component Architecture [1] specifications.

Unfortunately, no single RMI system provides all of those features. The problem of designing a "universal" RMI is difficult. The format of data and the protocol used to exchange it is a determining factor in the degree of interoperability among applications. The lack of a reliable, universally understood data-exchange format has long limited effective communication between heterogenous systems. In recent months, XML [11] has emerged as a standard for representing data in a platform-independent way. XML is essentially a tree-oriented data representation language that is simple to generate and parse. Also, HTTP has emerged as a simple, universally supported protocol for exchanging data over the Internet. HTTP requests/replies are readily passed through firewalls and handled securely, unlike the notoriously unsafe execution of arbitrary remote procedure calls (RPC) or RMI code. Thus, moving XML data via HTTP is an attractive way for distributed applications to communicate with each other. SOAP does precisely that. By expressing RPCs independent of platforms, it opens the possibility of implementing other architecture-specific protocols in SOAP. In particular, this makes SOAP attractive as an intermediary protocol into which other protocols can be easily translated – one reason why Microsoft has targeted SOAP as an entry mechanism to the COM world [3].

The same feature that makes SOAP attractive causes a potential performance problem: tagged data is sent as characters. By contrast, Nexus [17] is designed for high-performance communications and when layered with HPC++ [19] presents to the user a complete RMI system which can interoperate with Java [24]. However, Nexus has robustness problems and the failure of a global pointer can lead to complete deadlock of the distributed computation. Java RMI tends to be more robust, partly because of Java's exception and error-handling system. Java RMI is, however, a single-language protocol, and scientific computing relies on languages like Fortran90, C/C++, and Matlab as well.

This paper addresses the following questions:

- What is the raw performance of SOAP when used as a foundation for a RMI system?
- How does SOAP performance compare with that of Java RMI and Nexus?
- Where are the bottlenecks in SOAP performance? Which are removable through better implementations, and which are inherent in the protocol itself?
- Can a dynamic multi-protocol system be designed that achieves the benefits of several runtime systems?

This paper addresses these questions and shows that SOAP can be used to build a reliable, multi-protocol RMI system that can access desktop component technology like Microsoft COM and other non-Java software components. However, when additional performance is needed a multi-protocol approach allows a faster, more specialized protocol to be dynamically inserted to move data. Several efforts have been started to extend SOAP to have security and higher performance [2, 7] but at the cost of reducing its simplicity and universality. Many modern scientific computing systems now involve multiple languages, using the strengths of each where appropriate: Java for GUIs, Fortran for fast complex arithmetic, C/C++ for operating system interactions, Matlab for rapid prototyping, etc. Analogously, our work suggests that rather than try to extend a single communications subsystem to handle the wide range of scientific computing requirements, the more effective solution is to use multiple communication protocols.

2 Brief survey of RMI protocols

Although Java provides an interface for programming with sockets, typical applications require a higher-level protocol that can handle encoding and decoding of messages. Java RMI [25] is an API for remote method invocation – the invocation of a method in a remote object by a locally resident object. The polymorphism inherent in method calls makes the Java RMI API a more flexible alternative to RPC-based (Remote Procedure Call) APIs [30]. The actual class implementing the method can be dynamically loaded into the running application. The communication between the client and server is implemented by stub class on the client end and a skeleton class on the server end. The stub converts the arguments of the method invocation into a format used for transporting across the network to the remote object (this process is called serialization), and assembled together back again by the skeleton (deserialization) at the remote end. The result object is likewise serialized by the skeleton and deserialized by the stub.

Nexus RMI [6] is an implementation of the Java RMI API that uses Nexus [17] as the communication medium. Object serialization is achieved by adding public methods to serializable objects so that their private and protected fields can be accessed. Our experiments (see Figure 4, for example) show that Java RMI is at least four times faster than Nexus RMI. However the important feature of Nexus RMI is that it allows interoperability between Java and C++ [6].

Several recent efforts have addressed the problem of improving RMI performance. Philippsen et al. [28] have built drop-in replacements for JDK implementations that significantly improve performance. Thiruvathukal et al. [29] use explicit methods for serialization and deserialization in order to read and write an object's internal state. Manta [22] achieves impressive performance for RMI based on transparent extensions of Java for distributed environments. A native compiler generates efficient serial code and specialized serialization routines for argument classes. As a result these classes avoid run-time inspection.

3 SOAP

SOAP is an object-oriented, Internet-based protocol for exchanging information between applications in a distributed environment. Box [4] provides a good basic introduction to SOAP with some examples. SOAP is independent of the programming language, platform or transport mechanism used for the exchange. SOAP's interoperability arises from a simple syntax based on XML (Extensible Markup Language [11]). Although HTTP (Hypertext Transfer Protocol) is the most widely used transport layer for SOAP packets, which are XML documents, other protocols like SMTP or FTP can also be used. The SOAP message exchange model consists of one-way transmissions from sender to receiver which can be combined to be used as a request/response pattern. SOAP messages rely on XML Namespaces [8] and the XML Schema definition language [9]. The XML encoding makes SOAP messages simple to read and parseable by humans and machines alike, as testified by the plethora of XML parsers in various languages running on multiple platforms.

3.1 Related efforts

The universality and extensibility of XML facilitates the use of SOAP as a basis for building other higher-level services (e.g., protocols for service discovery, event subscription, message queuing, etc.). One example is Microsoft Corporation's Biztalk [12], a framework for secure document and message exchange based on XML and MIME (Multipurpose Internet Mail Extensions). Biztalk extends the SOAPv1.1 protocol and a BizTalk Document is a SOAPv1.1 message. The message body is a MIME document and the message header contains BizTalk-specific entries.

In scientific computing, XSIL (Extensible Scientific Interchange Language [31]) is an XML-based system that consists of a data format for describing scientific data and a mapping into the Java object structure. XSIL provides a core set of basic elements, (tables, arrays, streams etc.) that can be extended with user-defined data element types. A SOAP encoding-style could permit XSIL to be transported in SOAP messages.

3.2 Remote procedure calls in SOAP

Remote procedure calls in SOAP are essentially client-server interactions over HTTP where the request and response comply with SOAP encoding rules. The Request-URI (Universal Resource Identifier) in HTTP is typically used at the server end to map to a class or an object, but this is not mandated by SOAP. Additionally, the HTTP header `SOAPAction` specifies the interface name (a URI) and the name of the method to be called on the server. The SOAP message is an XML document whose root element, the Envelope, specifies the overall structure of the message, its intended recipient, and other attributes of the message. SOAP specifies a remote procedure call convention, which includes the representation and format to be used for calls and responses. A method call is modeled as a compound data element consisting of a sequence of fields (accessors), one for each parameter. A return structure consists of the return value as well as the out and in/out parameters. SOAP encoding rules specify the serialization for primitive and application-defined datatypes.

Figures 1 and 2 show the request and response structure of a remote procedure call transported as an HTTP request carrying a SOAP payload.

```
POST /Temperature HTTP/1.1
Host: www.temperature-service.com
Content-Type: text/xml
Content-Length: 357
SOAPAction: "http://weather.org/query#GetTemperature"

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetTemperature xmlns:m="http://weather.org/query">
       <longitude>39W</longitude>
       <latitude>62S</latitude>
    </m:GetTemperature>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1: Example of a SOAP request sent via HTTP.

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 343

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetTemperatureResponse xmlns:m="http://weather.org/query">
      <centigrade>28.4</centigrade>
    </m:GetTemperatureResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2: Example of a SOAP response received via HTTP.

SOAP allows hierarchically structured queries and responses, and specifies serialization of primitive string, numeric and date datatypes, and aggregates like arrays and vectors. Sparse arrays, and protocols for sending parts of them are also supported. New types may be defined using the `<complexType>` construct inside a schema definition.

Overall, SOAP provides many advantages. Unfortunately, its universality comes with a performance penalty: XML messages are textual and so the sizes of its messages are significantly larger than protocols which send binary data. Since a distinguishing characteristic

of scientific computation is the need to handle large data sets, the performance of SOAP relative to specialized protocols that can use binary representations is an important issue. The next section tests SOAP performance relative to other communication protocols.

4 Performance results

A central issue in any system that relies on multiple protocols is finding break-even points – to know when one method is preferable to another. In numerical computing such approaches are called *polyalgorithms* [20], and the breakeven points can often be specified in terms of a few parameters giving problem characteristics independent of the computing environment. In communication systems, however, the issue is significantly more complex because of dependence on hardware, networks, and software implementations. Particularly for SOAP which is undergoing rapid development (for example, currently there is no publicly available SOAP parser for C++), an extensive testing framework is required.

This section describes the framework used for performance tests, the tests performed and observations on the results. The framework, written in Java, executes test programs that are instrumented to automatically accumulate performance data over several runs in a common format suitable for visualization and other processing. Plots showing data for the tests performed are presented in the appendices. For each test, the mean value of accumulated runs is plotted together with error bars corresponding to one standard deviation in each direction.

Figure 3 summarizes the machines used in the tests. These machines range from typical workstations to high-end servers. All the UltraSPARC machines were running SunOS v 5.7; the Linux machines were running RedHat 6.2 with kernel version 2.2.16.

On the UltraSPARC's, JDK 1.2 Solaris VM (build Solaris_JDK_1.2.2_05a, native threads, sunwjit), and JDK 1.3 Standard Edition (build 1.3.0-beta_refresh) with Java HotSpot(TM) Client VM (build 1.3.0-beta_refresh, mixed mode) were used. On the Linux machines, JDK 1.2 Classic VM (build 1.2.2_006, green threads, nojit) and JDK 1.3 Standard Edition (build 1.3.0beta_refresh-b09) with Java HotSpot(TM) Client VM (build 1.3.0beta-b07, mixed mode) were used.

Java's `System.currentTimeMillis()` call was used for timing measurements. A high-resolution clock was used for some tests on UltraSPARC systems. The tests were divided into sets A, B, and C. Each set was executed on various combinations of machine configurations, hardware environments, and protocols.

Data types used The experiments measured the roundtrip time for sending and receiving a linked list object or an array of doubles between two machines. These two datatypes were chosen as representative for scientific applications. Arrays account for the vast majority of their data, and a linked list is challenging for serialization since it requires the RMI system to keep track of previous references to avoid infinite loops. Linked lists have characteristics common with data objects used in scientific computing, such as sparse matrices and trees that support adaptive mesh refinement and N-body simulations. Each node in the linked list was a simple object with the following data fields:

```
public class LinkEntry implements Serializable {
```

name	Arch.	CPU	Mem/Swap(GB)
local	Ultra-10	UltraSPARC-IIi, 440MHz	0.256/ 1.1
sparc10	Ultra-10	UltraSPARC-IIi, 400MHz	0.256/ 1.1
linux	PC	Pentium-III, 533MHz	0.256/ 0.5
linux	PC	Pentium-III, 533MHz	0.384/ 0.5
e10k	Enterprise 10000	Sixty UltraSPARC-II's, 400MHz	60/ 60
e10k	Enterprise 10000	Four UltraSPARC-II's, 400MHz	4/ 10
remote	Ultra-10	UltraSPARC-IIi, 440MHz	0.256/ 2

Figure 3: Specification of machines employed in experimentation

```
  public long llong;
  public short sshort;
  public int iint;
  public float ffloat;
  public double ddouble;
  public String sstring;
  public LinkEntry nextLink;
}
```

The linked list sizes were varied from 10 to 10,000 nodes, while array sizes ranged from 10 to 50,000 entries. Although scientific computing now involves arrays several orders of magnitude larger, these sizes sufficed for determining the relative RMI performances.

Protocol implementations The first RMI tested was Sun's native implementation using the JRMP (Java Remote Method Protocol) protocol. The second protocol tested was Nexus. Although Nexus RMI supports language-interoperability between Java and C++ and HPC++[6], tests were conducted just for Java.

The third protocol used was SOAP RMI, which is an early implementation of RMI based on nanoSOAP, our implementation of a simple SOAPv1.0 serialization and deserialization mechanism. SOAP RMI uses an XML-Schema specification of the server interface to generate the associated stubs and skeletons. A remote object reference is an HTTP URL along with information that uniquely identifies the instance. The stubs and skeletons do not directly interact with the SOAP implementation, but instead use a communication object which is an abstraction that helps hide the underlying implementation of SOAP. This design is useful as it allows run-time insertion of different SOAP implementations.

The tests were executed for only up to 10,000 objects. Memory limitations prevented larger test sizes; for example, in Sun RMI the recursion stack depth of around 1000 was a limiting factor. We were able to run bigger sizes only by adjusting the stack size option (-Xss) for the Java virtual machine, which can affect performance.

4.1 Test sets

Test set A: Overall performance Test A computes the total roundtrip time for sending and receiving aggregate objects. The parameters of the test include the type of the aggregate (linked list or array), size of the object (number of items in the aggregate), client-server pair, and protocol. The following client-server combinations were used: sparc10-sparc10, linux-linux, e10k-e10k, all three involving two machines of the given architecture connected by 100 Mb/s Ethernet LAN. The local-remote tests pair two UltraSPARC stations, one at Bloomington, Indiana and the other at Indianapolis, connected by a 155 Mb/s WAN. Appendices A and B give the full set of results for Test A, which are discussed below in Section 4.2.

Test set B: Raw performance and protocol overhead The second set of experiments measured serialization and deserialization performance of a linked list and an array of doubles using Sun, Nexus, Apache SOAP and nanoSOAP serialization and deserialization implementations. Apache SOAP [14] is a reference implementation of SOAP v1.1 based on IBM's SOAP4J implementation. The serialization and deserialization tests did not involve any communication and so factor out network vagaries. Tests were performed for all of the architectures. Appendices C and D contain figures showing the overall results for Test B.

Test set C: Gantt diagrams These tests measured the contribution of each phase in a single RMI invocation roundtrip that sends and receives a linked list of 1000 elements. A simple RMI example based on SOAP serialization and deserialization was instrumented to identify the time-critical phases in a single remote method invocation and response roundtrip, in part to identify any overlap or concurrency in the communication.

4.2 Observations

Figure 4 compares throughput for Sun RMI, Nexus RMI and SOAP RMI. The performance was also compared to a transfer of serialized array and linked list data over a raw socket connection. In general SOAP RMI is approximately ten times slower than Sun's implementation, not surprising considering the relative sizes of data that must be sent for the same object. Surprisingly, SOAP RMI outperforms Nexus RMI for small data sizes (Figures 11 and 12). For the linked list example, the cross-over point was around 10 nodes; for double arrays this was around 100 elements. Since SOAP RMI is consistently slower in the corresponding serialize/deserialize tests, this implies that the effect comes from buffering implementations and possibly from the cost of conversion to network representations. It also implies that using SOAP is not only acceptable for small messages such as control signals or small parameter exchanges between remote components, but that SOAP is actually preferrable in some cases.

Sun's native Java serialization and deserialization is closely tied to Java and not surprisingly, turns out to be the most efficient of the four. Nexus's serialization and deserialization protocols are also binary, like Sun's, but the serialization is designed to be interoperable with C/C++, which adds some overhead to its performance relative to Sun's native Java serialization. Apache SOAP uses DOM [10] for deserialization. It provides the ability to

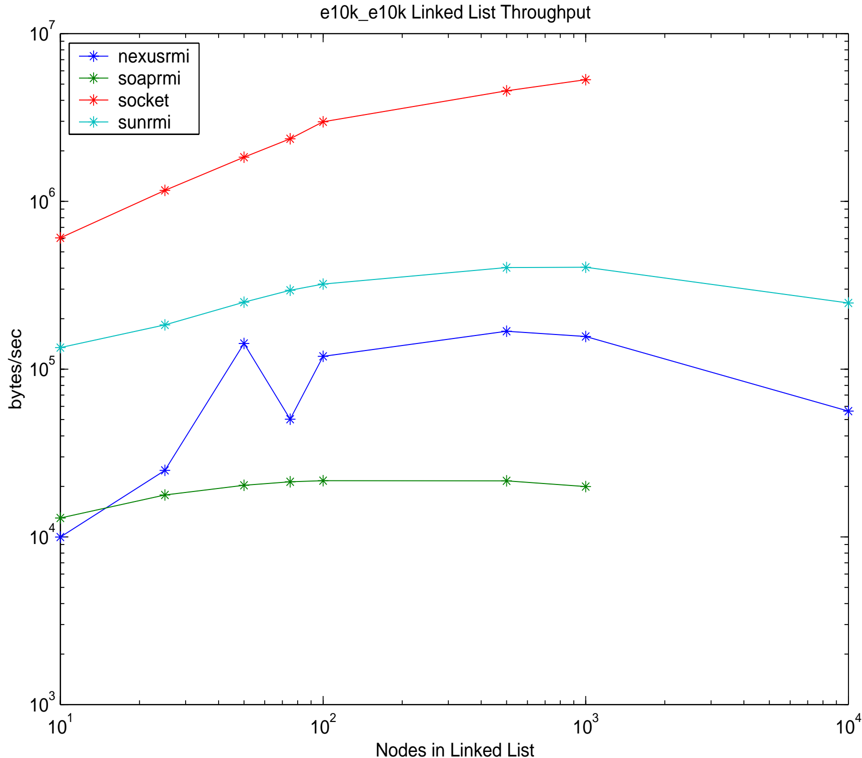

Figure 4: Roundtrip throughputs between E10Ks for linked list

plug-in different encoding-styles: Soapv1.1 Encoding, literal XML and XMI (XML Metadata interchange [21]). The nanoSOAP implementation is a fast, simple implementation of serialization and deserialization of Java into SOAPv1.0 encodings and it uses SAX [16] for deserialization. Figure 6 shows that the size of a serialized data types in SOAP is approximately ten times larger than in Sun native serialization. This increase in size is from the translation of binary data into text. For example, in Java, each double takes 8 bytes. The string representation in XML of a double with 16 digits of precision takes at least 16 characters (and so 16 bytes in UTF-8) in addition to the 17 bytes for the tags `<double>` and `</double>`. Thus, each double serialized into XML could take at least 33 bytes. If the data is transferred using Unicode, that estimate doubles. In any case, the overhead is at least a factor of four larger in the XML representation of a double array. Since SOAP uses XML for data representation, this overhead is intrinsic to the SOAP protocol and cannot be removed by choosing a better implementation.

Serializing Java objects into SOAP-encoded XML data takes approximately ten times more memory than the binary representation. Figure 5 compares serialization and deserialization throughputs for E10K. Sun's native serialization-deserialization is the fastest. Nexus's performance is comparable to Sun's. Serialization and deserialization speeds for SOAP-based (Apache SOAP and nanoSOAP) implementations are approximately 100 times slower and their throughtputs are also a 100 times lower.

Some additional observations can be made from the experiments:

- The most significant defect of using SOAP for RMI is performance; just sending the

8-byte double in XML, `<double> 3.141592653589793E+000 </double>`, requires 40 bytes of data. Determining the precise performance penalty is important for deciding when SOAP is appropriate. Figure 6 shows that SOAP's data representation size in general is about 10 times the size of binary representations.

- The asymptotic behavior between socket and Sun RMI (especially for the local-remote test) is similar (Figure 11). This indicates that Sun RMI imposes a minimal overhead for marshalling and unmarshalling of data.
- The nonmonotone behaviour of Nexus RMI (see Figures 11, or 12) appears consistently and is statisticially significant. Since it does not appear in the corresponding serialization tests, this implies it is an artifact of how Nexus RMI handles buffers or its on-wire data representation.
- There is a clear difference (a factor of 2-10) in performance between JIT-compiled and interpreted code for Linux (Figure 9). Serialization in particular has small tight loops that are readily amenable to compilation optimizations.

The Gantt diagrams from Test C in Figure 7 shows that costs of actual communication and data copying are considerably smaller than time spent on serialization and deserialization of XML encoded messages. Serialization and deserialization need to be the primary targets for improving performance of SOAP-based protocols. The graph data suggests at least a thirty percent improvement is possible if serialization and deserialization were pipelined. In more detail, the time taken for the roundtrip linked list example may be divided into the following segments:

- Serialization converts an object into its persistent state; SOAP uses XML as its serialization format. Java RMI serialization is approximately 100 times faster than nanoSOAP.
- Deserialization converts objects from their persistent state to their representation in memory. Deserialization in SOAP involves parsing the XML representation of an object and instantiating the object using reflection. In Java RMI deserialization the class structure of the object being deserialized is already known. On the other hand, in SOAP deserialization the class structure is learned as the XML is parsed. This coupled with the already large size of the XML representation of the serialized object makes the SOAP deserialization considerably less efficient. The Figures in Appendices C and D show this dichotomy consistently.
- Buffer copying: Ideally, any distributed object protocol should ensure that there is no copying of buffers from the network layer to the runtime system (i.e., zero-copy protocol). However, for transporting SOAP over HTTP, the serializer output needs to be copied into a buffer before sending it on the wire if the length of the stream is to be sent as an HTTP header. The cost of copying is small and could be eliminated if the content-length header is not sent.
- Network: RMI involves sending serialized representations of objects over a network. The time taken for this is directly proportional to the size of the serialized representation for the low-latency networks and large objects used in the testing.

5 Multi-protocol design

The experiments show that SOAP has a significant performance penalty. Some of this is inherent – SOAP must send larger amounts of data, and that cannot be reduced without changing the protocol. Some of it may be reduced by better implementation of phases like deserialization, although the constraint of handling arbitrary complex data objects will bound the performance enhancements. Furthermore, performance is only one capability to consider in selecting communication protocols. Java RMI has the advantage of allowing users to quickly incorporate capabilities like database interfaces, compression, encryption, and visualization, but Java RMI is limited to a single language. Earlier work [13] showed that Nexus can provide high-speed data transfers between two C++ components, but suffers from robustness problems for long-running applications. Java RMI has outstanding performance, but the limit on its recursion stack depth prevents it from being used on arbitrarily-sized data unless the application-level user does chunking – a significant burden to place on an applications scientist.

In distributed scientific computing component systems, applications are started up on remote machines and wired together dynamically while some parts are running. Components may be migrated to other machines during runtime based on resource availability and network load. In this context the choice of protocol depends on dynamically changing factors: size of the data, security policies, dynamic reconfiguration of component wiring, quality of service requirements, etc. A component may be connected to several components at any given time, each possibly using a different protocol. In this case, universal connectivity is important. As an example, the Common Component Architecture Toolkit (CCAT) system [5] is a framework that connects scientific computing components in C, C++, Java, and Fortran. Components are dynamically created, connected, disconnected, and terminated. A component also can be a remote instrument like an X-ray crystallography data collector, or distributed databases. Messages between components range from short packets for lifecycle maintenance or setting internal parameters up to gigabyte data transfers. Components can represent extremely long-running or expensive services, for which robustness is critical.

5.1 Design of a Multi-Protocol RMI System

A multi-protocol RMI system should have the following properties:

- Provide a common-denominator protocol.
- Architect a common RMI API to different protocols.
- Provide the user with the ability to dynamically switch between protocols.
- Enable dynamic discovery of protocols.
- Implement reliable RMI in a heterogenous environment through a failsafe mechanism.
- Use meta-information (XML Schemas) to define component interfaces.
- Automatic generation of stubs and skeletons from component meta-information.
- Allow high-speed, large data transfers.

SOAP RMI addresses many of these criteria except the last, and as the experiments have shown, for small data sizes SOAP can be faster than Nexus. Having character-based

messages, SOAP RMI provides a central, universally accessible protocol that can be readily translated into other protocols. Combined with XML Schemas it is a *lingua franca* for meta-information.

Figure 8 shows the top level design of a proposed multi-protocol RMI system. Our current testbed supports Java RMI, Nexus RMI and Soap RMI. SOAP RMI is the default protocol that all clients and servers are expected to support. However the testbed is designed to accommodate any RMI system. RMI consists of two parts: publication and discovery of remote services (the *bind* and *lookup* operations), and data transfer between the client and server via invocation of the remote method. The testbed system requires a registry for each supported protocol. However, clients need to be aware of the existence of just one registry, the SOAP RMI registry, as SOAP RMI is the default protocol.

A client can specify the protocols it supports to the protocol-suite. The protocol-suite maintains two lists: the first list has the protocols that the server supports and the second has the protocols that the client supports. When a client invokes a method on a remote reference, the method is actually invoked on the meta-stub. The meta-stub queries the protocol-suite to decide which protocol to use. The protocol-suite does a simple match between the client-list and the server-list to determine the protocol to be used for the current method invocation. The protocol can therefore be switched on a per-remote-method call basis. The client can however override this mechanism and fix the protocol to be used before the method invocation. The implementation provides mechanisms so that any policy can be used by the client for protocol selection.

6 Conclusions

We analysed the performance of SOAP and the role it can play in distributed object component systems. As expected, SOAP RMI is usually much slower than Java RMI and Nexus RMI, usually by a factor of about ten. Surprisingly, it can be faster than the high-performance protocol Nexus RMI for small messages. Since its serialization and deserialization times are always larger than Nexus's, the faster speed is attributable to what happens between the client and server, including conversions to network representations. In spite of this the XML messages SOAP uses are inherently unsuitable for bulk data transfers.

Although the performance break-even point occurs for small messages, SOAP's interoperability across heterogeneous environments makes it a valuable ingredient for a multi-protocol environment. Human readability of SOAP packets makes SOAP a useful protocol during development and debugging. Since SOAP's simplicity lends itself to robust implementations, it should be used as a failsafe mechanism, or as a protocol for exception management. In general, because no single protocol is suitable for all situations in scientific computing, a multi-protocol system is needed that can benefit from the strengths of each one.

This multi-protocol design is being integrated into the Common Component Architecture Toolkit framework. The first step is a C++ implementation of SOAP RMI so that it directly interoperates with Java and C++ based components. Fortran components use existing Fortran-C++ interfaces to access messages, while Matlab uses its Java interface.

7 Acknowledgements

This work was supported in part by DARPA, the DOE2000 project, the NCSA Alliance, and the NSF NGS project Kenneth Chiu helped with statistical evaluation of raw benchmark data and shared his insight about performance issues on Solaris.

References

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[2] John J. Barton and Satish Thatte. Soap messages with attachments, July 7, 2000. http://static.userland.com/weblogsCom/ gems/soapweblogscom/soapMessagesWithAttachments.html.

[3] Don Box. Lessons from the component wars: An xml manifesto. http://msdn.microsoft.com/workshop/xml/articles/xmlmanifesto.asp.

[4] Don Box. A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages, visited 07-26-00. http://msdn.microsoft.com/msdnmag/issues/0300/soap/soap.asp.

[5] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing Conference, Pittsburgh*, August 1-4 2000.

[6] 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 and Experience*, 1998. Presented at 1998 ACM Workshop on Java for High-Performance Network Computing.

[7] David Burdett. Requirements for xml messaging, visited 07-01-00. http://www.ietf.org/internet-drafts/ /draft-ietf-trade-xmlmsg-requirements-00.txt.

[8] World Wide Web Consortium. Namespaces in XML, 1-14-99. http://www.w3.org/TR/REC-xml-names/.

[9] World Wide Web Consortium. XML Schema (Parts 1 and 2), 4-7-00. http://www.w3.org/TR/xmlschema-1/.

[10] World Wide Web consortium. Document object model, visited 7-15-99. http://www.w3c.org/DOM.

[11] World Wide Web Consortium. XML, visited 7-20-99. http://www.xml.org.

[12] Microsoft Corporation. BizTalk Framework 2.0 Draft: Document and Message Specification, visited 07-01-00. http://msdn.microsoft.com/xml/articles/biztalk/biztalkfwv2draft.asp.

[13] S. Diwan and D. Gannon. Adaptive resource utilization and remote access capabilities in high-performance distributed systems: The Open HPC++ approach. *Journal of Cluster Computing*, 2000.

[14] J. M. Duftler, S. Weerawarana, and F. Curbera. SOAP For Java, visited 7-01-00. http://www.alphaworks.ibm.com/tech/soap4f.

[15] D. Box et al. Simple Object Access Protocol 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[16] D. Megginson et al. Sax 2.0: The simple api for xml, visited 07-01-00. www.megginson.com/SAX/.

[17] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[18] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[19] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter 3 HPC++ and the HPC++Lib Toolkit. Springer-Verlag, 1997.

[20] E.N. Houstis, J.R. Rice, and R. Vichnevetsky. *Intelligent Mathematical Software Systems*. North-Holland, 1990. Proceedings of the first IMACS/IFAC International Conference on Expert Systems for Numerical Computing, Purdue University 5-7 December 1988.

[21] IBM. XML Metadata Interchange, visited 07-15-00. http://www-4.ibm.com/software/ad/standards/xmi.html.

[22] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. *In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 173–182, May 1999.

[23] Microsoft. COM, visited 4-1-2000. http://www.microsoft.com/com.

[24] SUN Microsystems. The Java Programming Language. http://java.sun.com/.

[25] Sun Microsystems. Java Remote Method Invocation, visited 07-01-00. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[26] SUN Microsystems. Java Beans, visited 4-15-00. http://java.sun.com/beans/.

[27] OMG. Corba Component Model, visited 1-11-2000. http://www.omg.org/cgi-bin/doc?orbos/97-06-12.

[28] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *to appear in Concurrency: Practice and Experience*, 2000.

[29] George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience, 10(11-13):911-926*, 1998.

[30] J. Waldo. Remote Procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, 1998.

[31] Roy Williams. XSIL: Java/XML for Scientific Data, 7-00. http://www.cacr.caltech.edu/SDA/xsil.

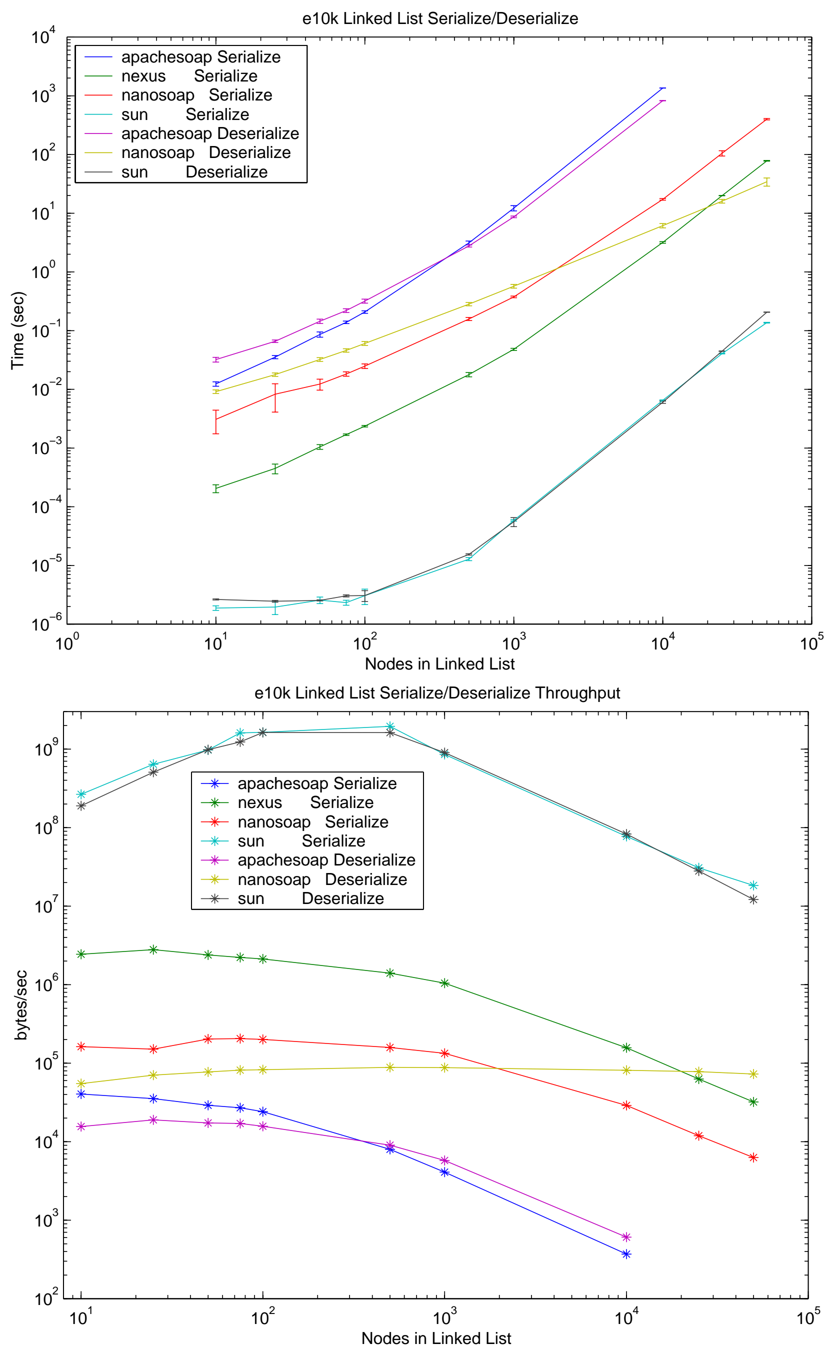

Figure 5: Serialization-deserialization for linked list on E10K (top: speed, bottom: throughput).

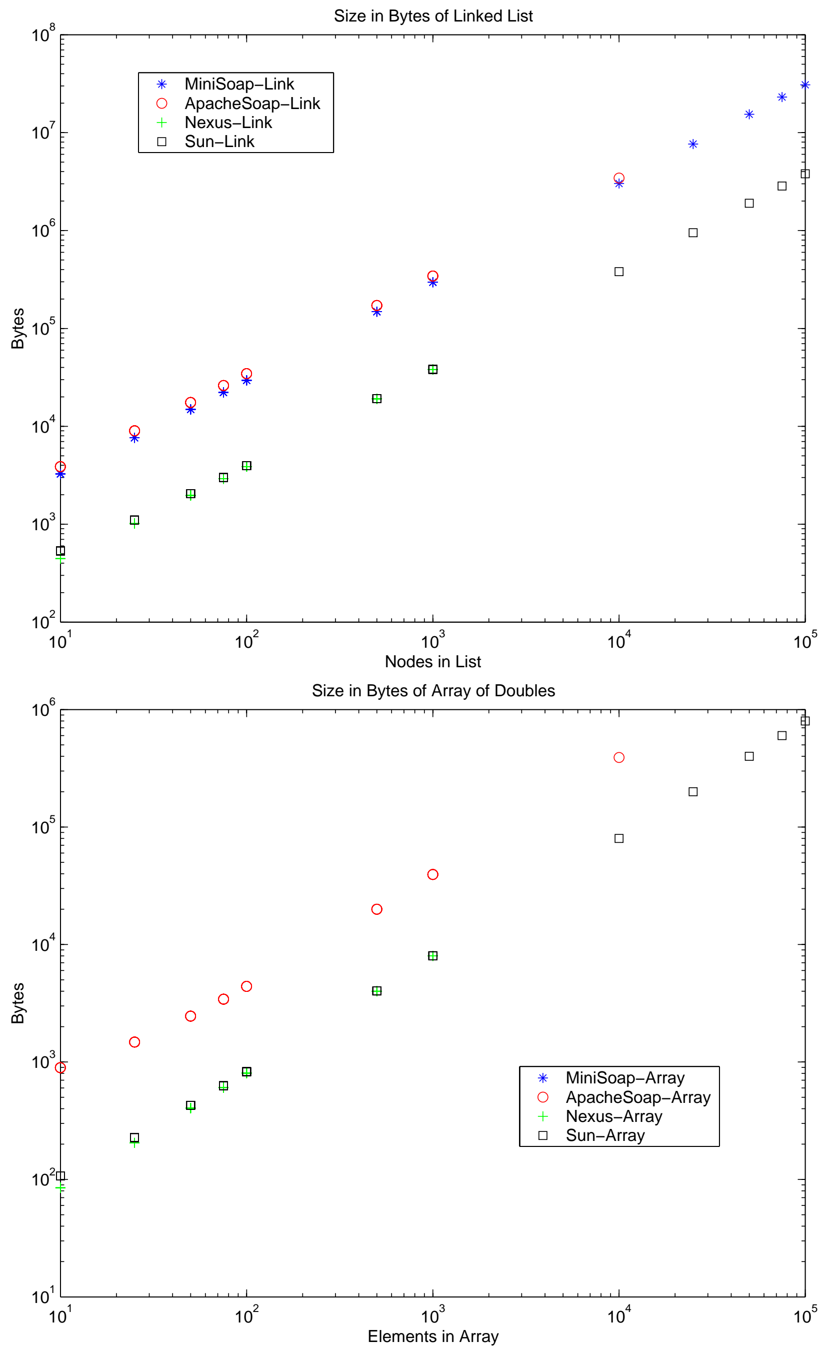

Figure 6: Serialization-deserialization sizes for linked list (top) and array (bottom) on E10K.

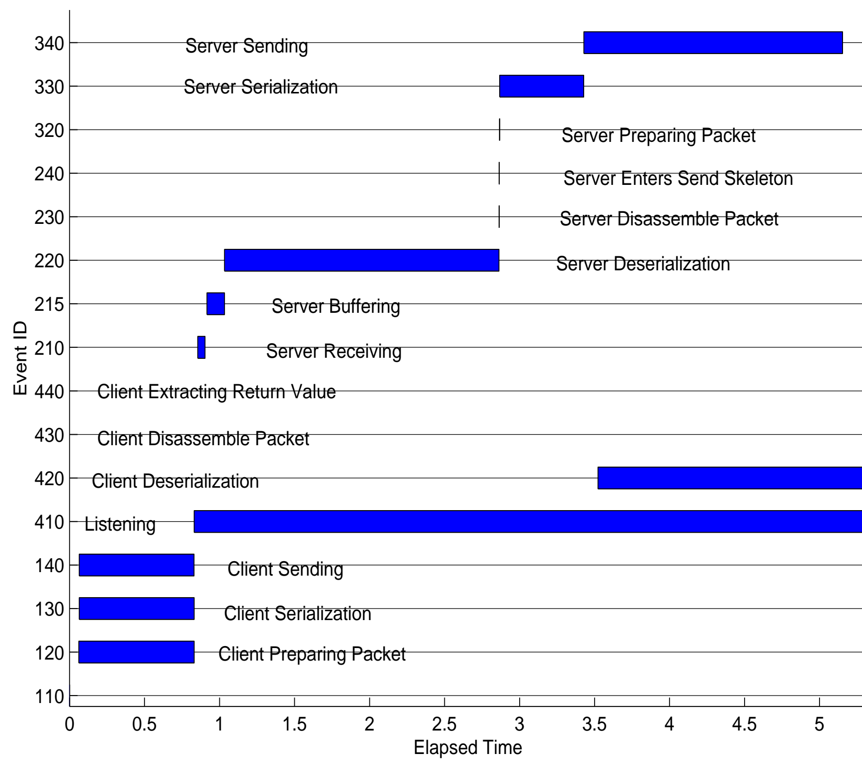

Figure 7: Gantt diagram for roundtrip for linked list of size 1000 on sparc10

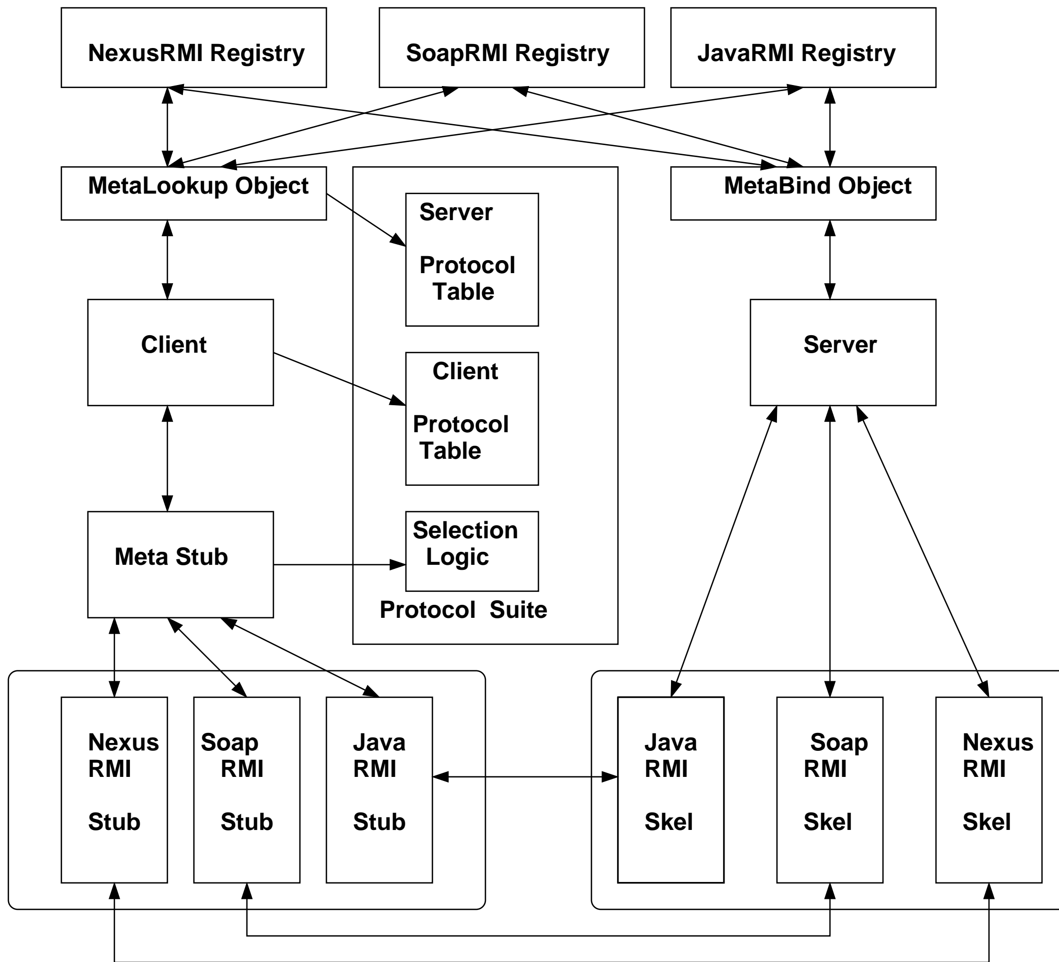

Figure 8: Multi-protocol design

A Overall RoundTrip Time Plots

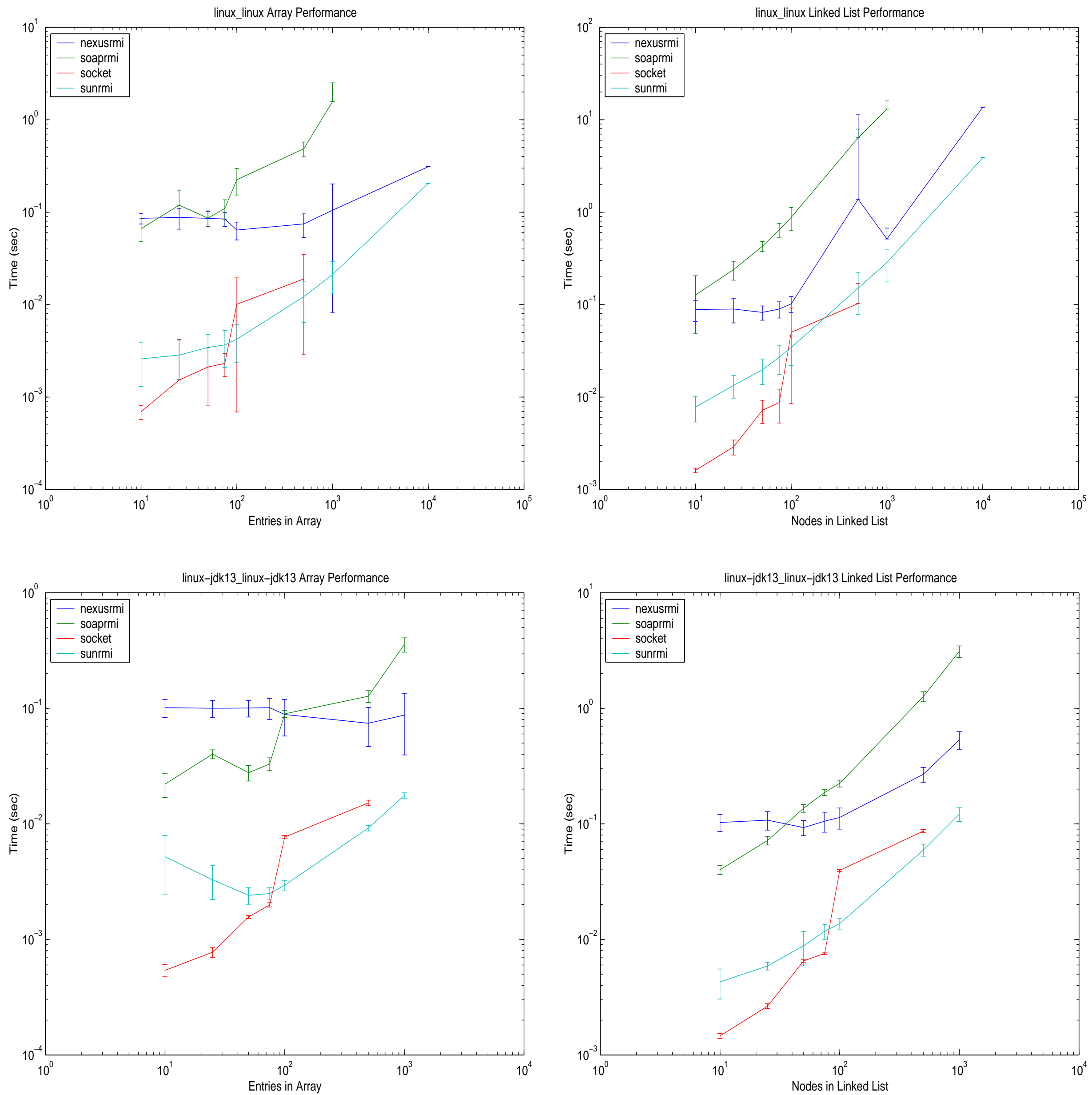

Figure 9: Roundtrip times from Linux to Linux on a LAN.Left side: Arrays; Right side: Linked Lists; Top: JDK1.2; Bottom: JDK1.3.

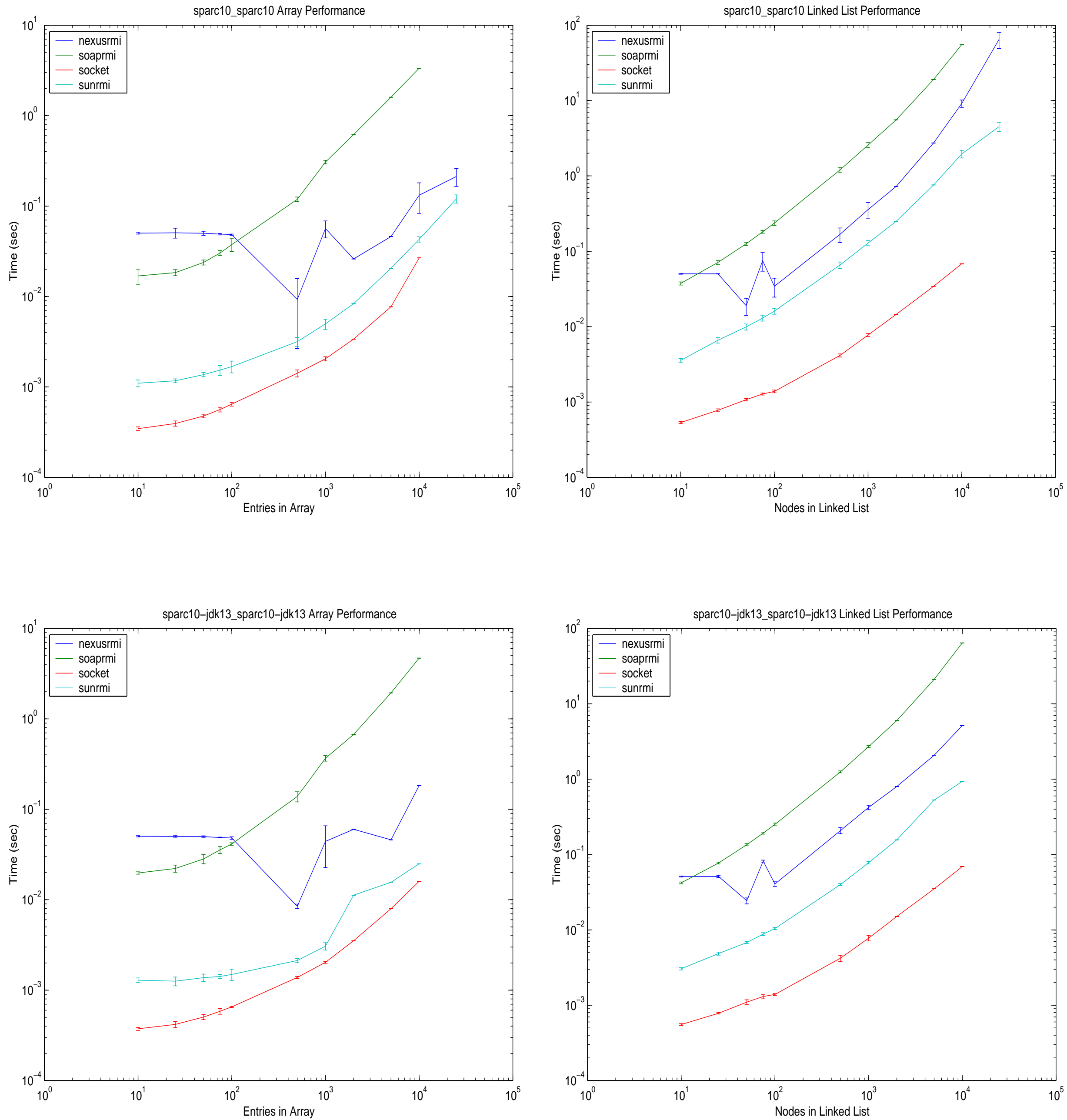

Figure 10: Roundtrip times from sparc10 to sparc10 on a LAN.Left side: Arrays; Right side: Linked Lists; Top: JDK1.2; Bottom: JDK1.3.

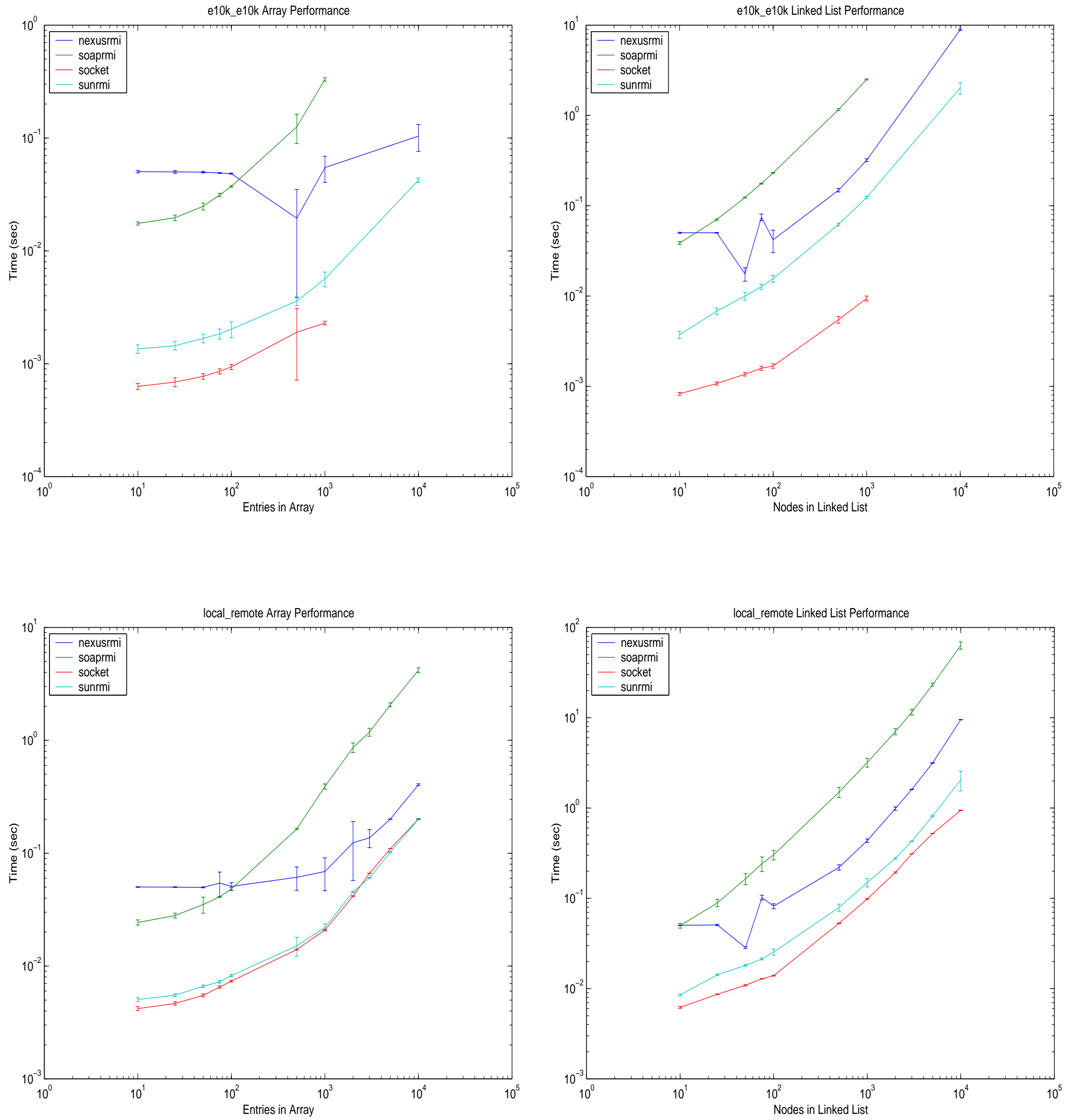

Figure 11: Roundtrip times between Sun E10K's on a LAN (top)and remote Sparc's (bottom). Left side: Arrays; Right side: Linked Lists.

B Overall throughput plots

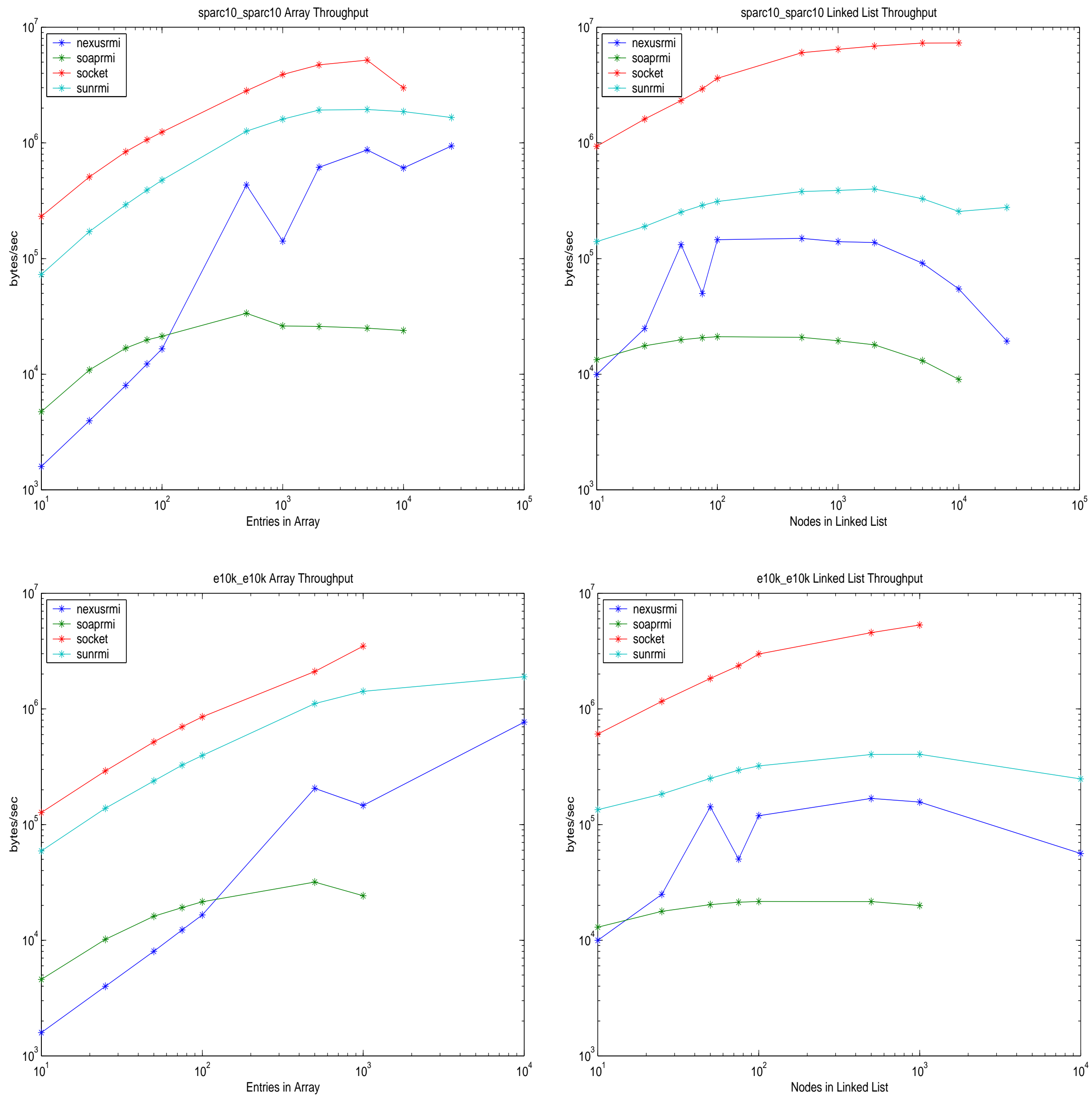

Figure 12: Roundtrip throughputs between Sparc 10's on LAN (top) and E10K's (bottom), for arrays (left) and linked lists (right).

C Serialize/Deserialize time plots

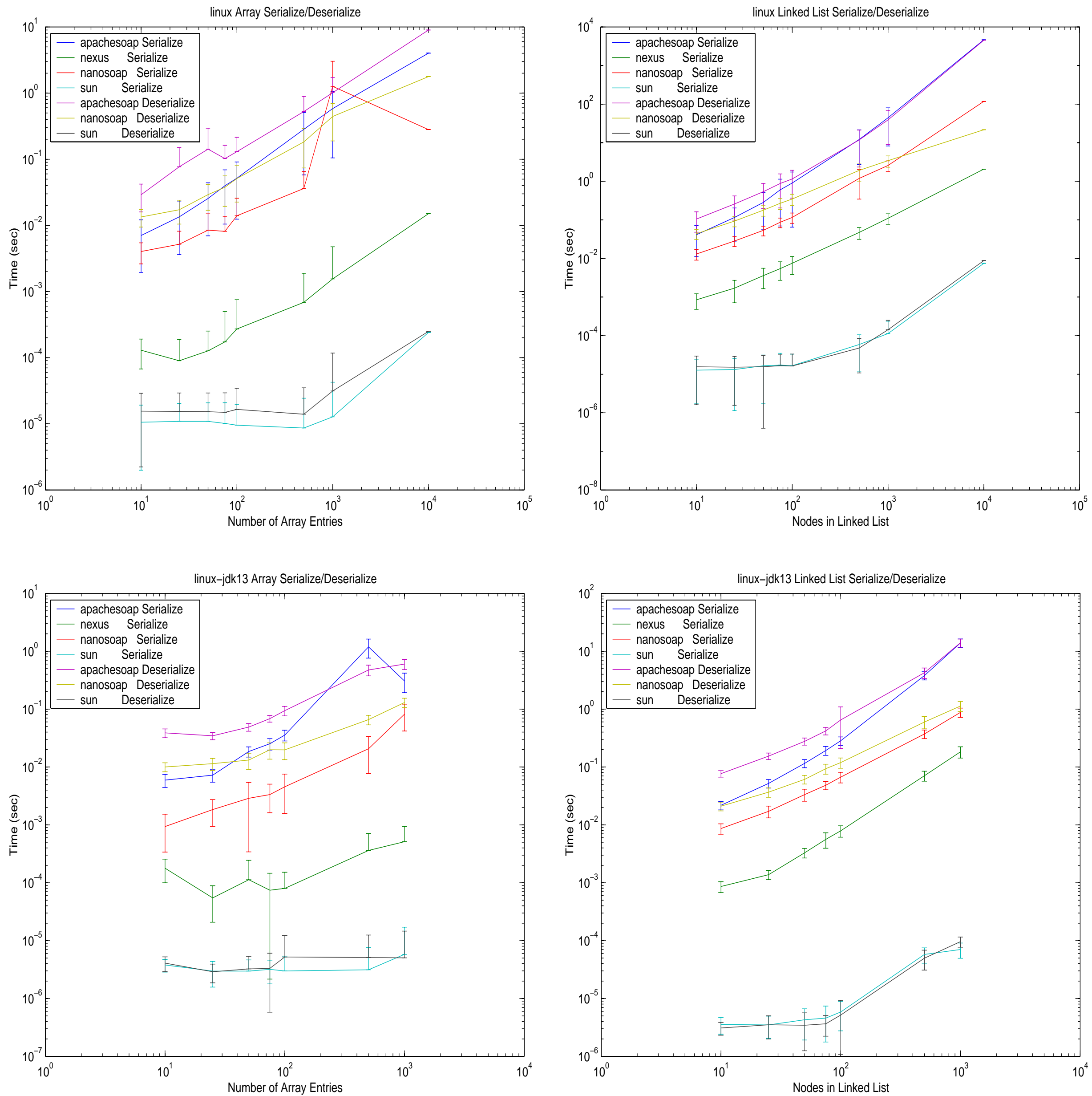

Figure 13: Serialization-deserialization times for Linux machines using JDK1.2 (top) and JDK1.3 (bottom), for arrays (left) and linked lists (right).

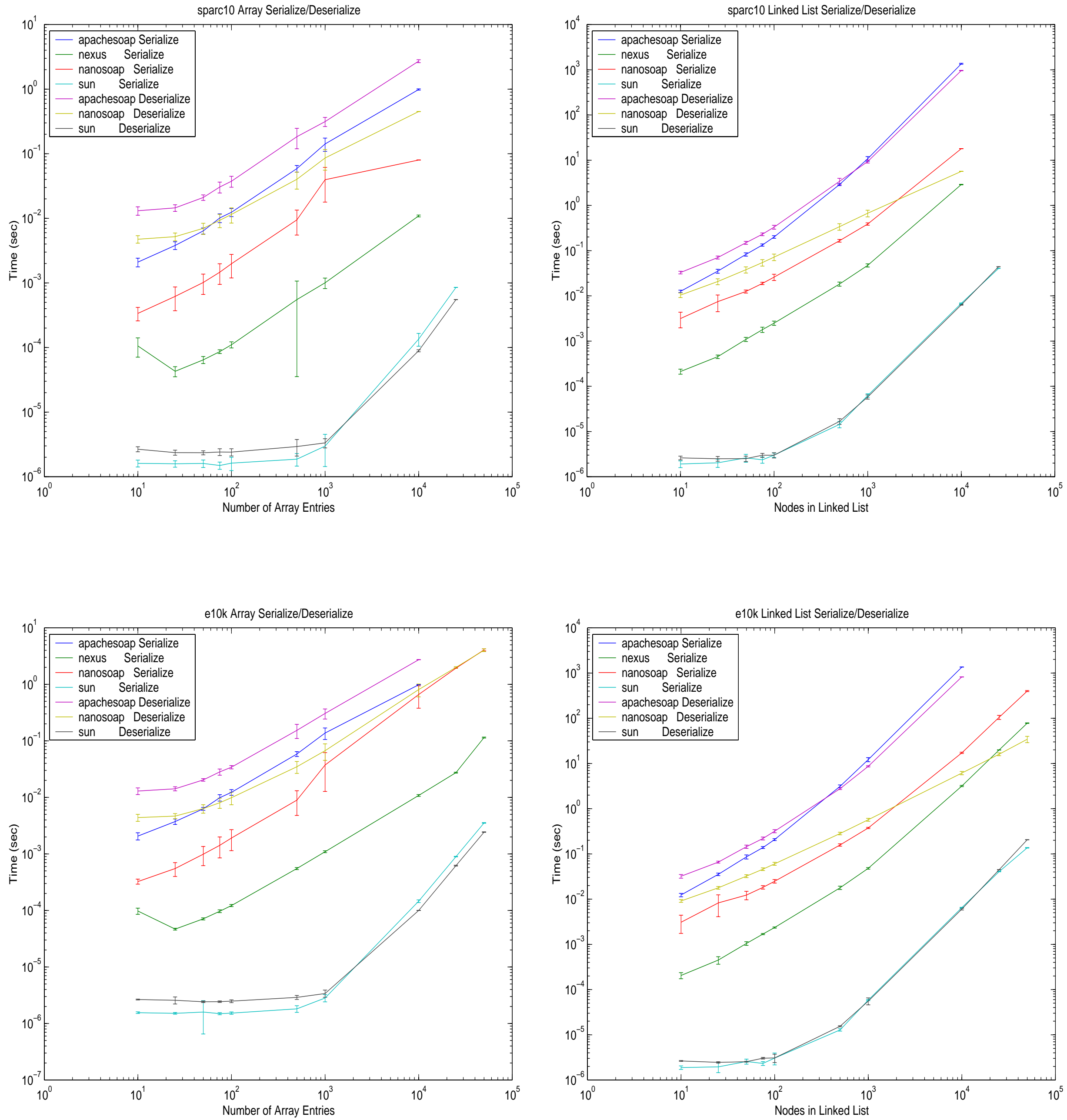

Figure 14: Serialization-deserialization times for Sparc (top) and E10000 (bottom), for arrays (left) and linked lists (right).

D Serialize-deserialize throughput plots

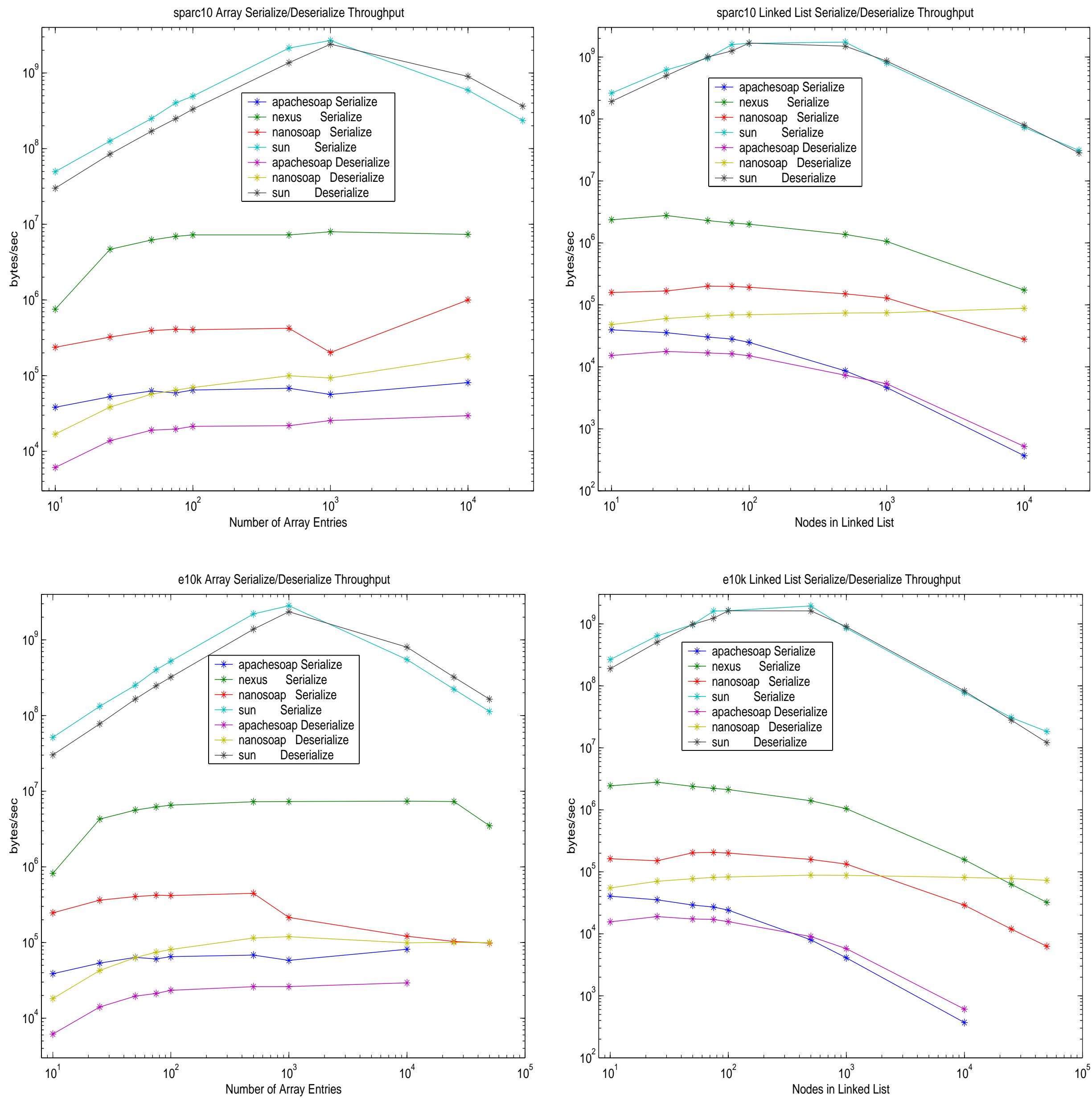

Figure 15: Serialization-deserialization throughputs for Sparc (top) and E10000 (bottom), for arrays (left) and linked lists (right).