A Software Architecture for Zero-Copy RPC in Java

Chi-Chao Chang and Thorsten von Eicken
Department of Computer Science
Cornell University

{chichao,tve}@cs.cornell.edu

Abstract

RPC has established itself as one of the more powerful communication paradigms for distributed computing. In recent years, object-oriented languages have impacted RPC semantics, with a number of variants providing remote method invocation and various forms of distributed object systems. At the same time, performance has changed little with the bottleneck being the network transport, in particular the in-kernel protocol implementations.

This paper describes J-RPC, an RPC architecture that leverages user-level network interfaces (UNI) to circumvent the kernel on the critical path. It describes how the wire format and the RPC system can be engineered to allow zero-copy reception of Java objects and zero-copy transmission of arrays. All objects received are fully type-checked and can be directly used by the receiving program. The design is connection-oriented for performance and leverages the JVM's garbage collector when managing receive buffers. An implementation built from an off-the-shelf JVM and a commercial UNI is used to evaluate the architecture and the tradeoffs of type-safe, zero-copy data marshaling.

1 Introduction

Since the conception of Remote Procedure Calls (RPC) [BN84], OS researchers have focused on both the RPC performance and programming paradigms that make RPC flexible, easy to use, and transparent. In the early years, the emphasis was on performance: first, between machines across the network, and then between processes on the same machine [JZ91, SB90, BAL+90, TRS+91, OCD+88]. At the time, the major bottlenecks were the slow network fabrics and the presence of the OS in the critical path (e.g. the system call overheads, in-kernel protocol implementations, and interrupt handling). The number of data copies was successfully reduced in the local RPC case. However, to do the same across the network required further research in network technology and in providing protected, direct user-level access to network interfaces. Meanwhile, with the increasing popularity of object-oriented and type-safe languages, focus shifted from performance to the semantics, flexibility, and ease of use of various distributed computing and object paradigms [BNO+93, OCD+88, Jav-a, EE98a, SGH+89, LCJ+87].

The advances in network fabric and interface technology in recent years have again made the performance of RPC an interesting issue. The introduction of low-cost networks with over 100Mbps of bandwidth and user-level network interfaces (UNI) that remove the kernel from the critical path have eliminated the perennial bottleneck in communication for commodity machines [vEBB+95, DBC+98, PLC95, CMC98]. Because NIs can directly access user-level buffers, highly-tuned implementations of RPC can customize the presentation layer to avoid any copying and achieve good performance. However, to our knowledge, such fast implementations [SPS98, BF96] currently only support untyped languages such as C and shallow data structures.

Type-safe languages such as Java add another dimension to the performance of RPC. RPC between Java programs must satisfy the type safety properties assumed by the language run-time (this is a general issue with safe languages). Enforcing this type-safety requires additional operations (checks) in the RPC critical path. In addition, passing arbitrary Java objects as RPC arguments requires full support for linked data structures. Finally, automatic memory management in safe languages calls for a careful integration between RPC buffers and Java's garbage-collected heap.

Section 3 presents an interface-based RPC model with simple calling semantics for Java that serves as a framework for experimenting with zero-copy data transfer. It offers the traditional RPC abstraction for Java

without the features (transactions, object migration, shared objects) that introduce substantial complexity into the system. Section 4 presents a novel wire protocol for typed objects that enables zero-copy data transfer and shows how arbitrarily complex and linked objects can be marshaled and unmarshaled efficiently in a type-safe manner. The key insight to achieving zero-copy is to preserve the representation of the object "on the wire" as much as possible, and to integrate the RPC buffers into Java's garbage-collected heap. Type-safety is achieved by incorporating typing information in the wire format in a way that enables efficient checking, that preserves the in-memory object layout, and that minimizes memory fragmentation at the receiving end. Linked objects are marshaled using portable pointers that are converted to real pointers in-place at the receiving end.

Section 5 describes a software architecture for RPC in Java on top of a UNI that provides efficient system support for the zero-copy protocol. It also relies heavily on the remote DMA support provided by state-of-the-art UNIs. The key design decision is to adopt a connection-oriented approach, which results in fast control transfer, efficient sender-managed flow control, and lock-free implementations of the protocol's marshaling and unmarshaling algorithms. These implementations can be done mostly in native code for high performance while providing the necessary API in Java for the stub generator.

It might seem paradoxical to try to push the RPC performance in a slow, safe-language system such as Java. Indeed, the safety of Java comes with a cost: slower overall execution mainly because of immature JIT compilers and garbage collection. However, safe languages need not be slow and their performance is likely to improve. While the slowness of Java might still be the real bottleneck in RPC-based distributed computing, the research presented attempts to map the fundamental limitations and performance tradeoffs in a zero-copy RPC system for safe languages.

The RPC implementation for Java discussed in this paper uses Microsoft's off-the-shelf Java Virtual Machine and is layered on top of a commercial UNI (GigaNet's GNN1000 VIA interface) [GNN]. We evaluate its performance with respect to the raw UNI and other off-the-shelf RPC systems. The results presented in Section 6 show that the system achieves a null RPC round-trip latency of 18.4µs which is 2µs above the raw UNI performance. The effective bandwidth for 8KByte arrays is 61 MB/s, about 75% of the raw bandwidth. We analyze the tradeoffs of zero-copy data marshaling and evaluate the behavior of the receive-buffers using traces of an extensible web server based on RPC. The paper concludes with a discussion of the lessons learned in designing an RPC system on top of a state-of-the-art UNI and using off-the-shelf JVMs.

2 Background

2.1 User-level Network Interfaces

The goal of UNIs is to remove the kernel from the critical path of communication while providing protected access to the network interface. As a result communication latencies are low, bandwidth are high even for small-sized messages, and applications have complete protocol flexibility. The latter allows the communication layers used by each process to be tailored to its demands so that advanced protocol design techniques such as Application Level Framing and Integrated Protocol Layer can be applied.

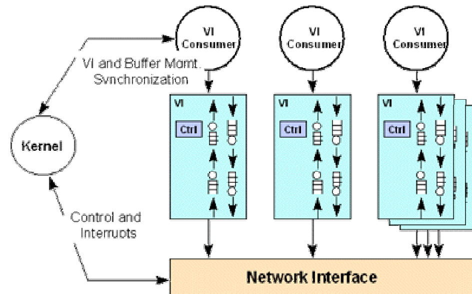


Figure 1: VIA Queue Architecture. Each process can open multiple virtual interfaces (VIs), each with its own send and receive queues. The kernel is involved in setting-up the VIs, mapping buffers, and handling interrupts.

The UNI used here is an implementation of the Virtual Interface Architecture (VIA) [VIA97], a standard proposed by Intel, Microsoft, and Compaq that combines basic send and receive operations in a UNI with remote memory transfers. Processes open virtual interfaces (VIs) which represent handles onto the network (see Figure 1). Each VI has two associated queues: a send queue and a receive queue which are implemented as linked lists of message descriptors, each of which points to one or multiple buffer descriptors. To send a message an application adds a new message descriptor to the end of the send queue. After transmitting the message, VIA sets a completion bit in the descriptor and the application eventually dequeues the descriptor when it reaches the head of the queue. For reception, the application adds

descriptors for free buffers to the end of the receive queue which VIA fills as messages arrive. VIA also supports remote direct memory access (RDMA) operations. A RDMA send descriptor specifies a virtual address at the remote end to which data will be written (RDMA-write) or from which data will be read (RDMA-read). Completion of RDMA-read operations may not respect the FIFO order imposed by the send queue. In addition, RDMA-reads do not consume receive descriptors at the remote end while RDMA-writes can if explicitly asked to. RDMA requires that the sender and receive exchange information about the virtual address prior to communication. VIA guarantees in-order delivery for send and RDMA-write operations.

In this paper, we assume that the underlying UNI architecture provides reliable communication, in-order delivery, and support for RDMA-writes.

2.2 Java

Java is a type-safe, object-oriented language that supports multi-threading and relies on a garbage collector. It offers two language features that are leveraged by the J-RPC model: *multiple interface extensions* (also known as sub-typing) and *abstraction*. Java distinguishes between regular classes that can be instantiated into objects and interface classes that only define a set of method signatures. Multiple interface extension means that a Java interface class can combine several other interface classes by extending them. This feature is useful in RPC systems because it can be used to support interface versioning — typically, an interface is upgraded by extending an older interface with new methods. Abstraction allows a Java class to be detached from the interface class it implements, leading to separate interface and implementation hierarchies. This means that a Java interface need not expose the implementation details to be inherited by interfaces that inherit from it. In addition, a Java class can extend other classes and implement multiple interfaces.

Besides these two features, we use the notion of a class *fingerprint* in order to allow type-checking across JVMs. A fingerprint is a 64-bit checksum with the property that with very high probability two classes have the same fingerprint only if they are structurally equivalent. Class fingerprints are obtained by calling `GetSerialVersionUID` provided by the Java Object Serialization API[1]. The rest of the paper uses the terms *fingerprint* and *type-descriptor* interchangeably.

3 The J-RPC Model

RPC models designed for object-oriented languages vary in how services or objects are exported from one program to another. This section describes the binding model used in the RPC system for Java presented in this paper, called J-RPC. J-RPC leverages the interface construct of Java and is entirely *interface-based*. This means that a program that wishes to export a service defines a Java remote interface (a Java interface that extends a special interface called `Remote`) which describes the methods being exported. The implementation of the service is provided by objects that implement the remote interface. The key point is that a program exports a specific interface implemented by an object and not all interfaces or all public methods of that object.

3.1 Binding

The example illustrated in Figure 2 shows two remote interfaces, iA_0 and iA_1, each implementing one method, `f` and `g` respectively, and a class `A` that implements both interfaces. The server code fragment creates an instance of `A` and exports its iA_0 interface with "IA0" as the service name. Exporting the instance of `A` creates a stub object that implements iA_0. The client imports "IA0" from the server's machine, which creates a proxy instance that implements iA_0 only (not iA_1) and establishes a binding with the stub, as seen in Figure 2b. Proxy and stub classes loaded by the Java class loader are registered under the interfaces they implement. In other words, the RPC system maintains a mapping between the type-descriptor of iA_0 and the proxy in the client side, and a mapping between the type-descriptor of iA_0 and the stub in the server side.

---

[1] Unfortunately, off-the-shelf VMs compute the checksum with different hashing algorithms. For example, for an identical class, the fingerprints produced by Microsoft's and Sun's VM are different.

```
interface iA0 extends Remote {
  void f(A a);
}
interface iA1 extends Remote {
  void g(iA0 a);
}
public class A implements iA0, iA1 {
  public void f(A a) { . . . }
  public void g(iA0 a) { . . . }
}
public class Server {
  . . .
  A a = new A();
  RPC.export(a, "iA0", "IA0");
  . . . }
public class Client {
  iA0 a0 = (iA0) RPC.import(
    "IA0", "128.84.223.121");
  . . . }
```

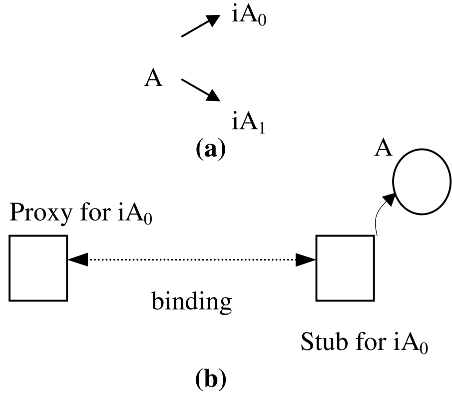


Figure 2. **(a)** Java class hierarchy for pseudo-code **(b)** J-RPC binding

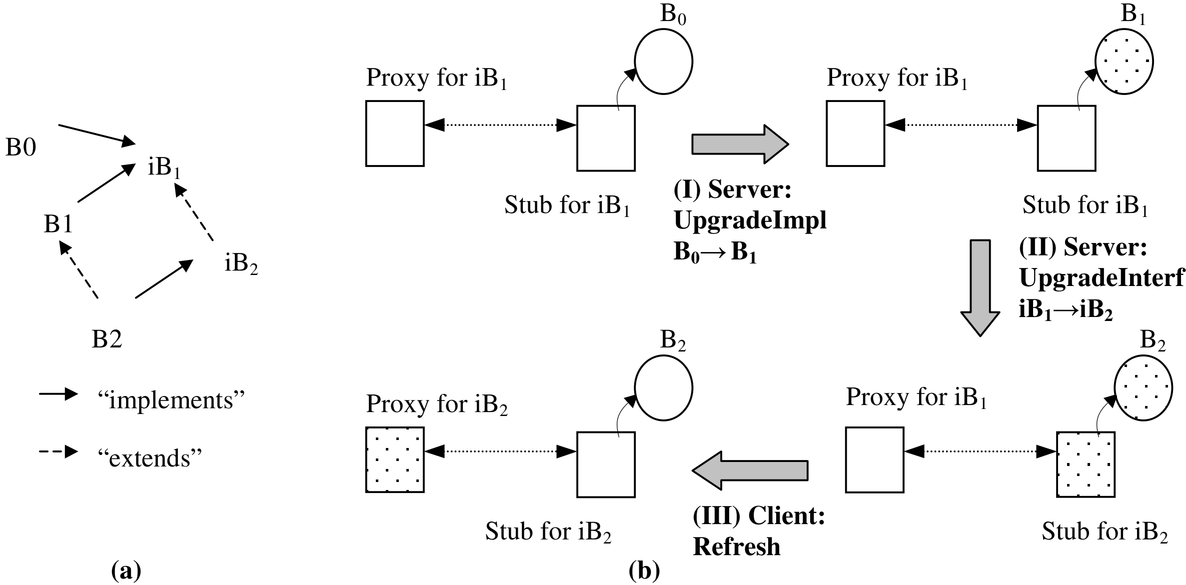


Figure 3. **(a)** A typical Java class hierarchy due to versioning **(b)** How versioning affects J-RPC bindings

3.2 Call Semantics

The argument passing semantics are dictated by the signature of the remote method. An argument is passed by copy if its signature type is a primitive type or a Java class and is passed by reference if its signature type is a remote interface. In the class-copy case, all transitively reachable objects are passed across the network (using the same by-copy/by-reference rule) except for objects referenced by *transient* fields. When passing a remote interface, the wire representation of the interface is transferred across the network and transformed into a proxy on the receiving end. For example, a remote invocation of `g` in Figure 2 would pass the wire representation of the remote interface iA_0 to the callee, whereas remotely calling `f` would pass a copy of an object of class `A` to the callee. In the former, the callee of the `g` gets a proxy that

implements $\mathtt{iA_0}$. In the latter, the callee of `f` simply gets an instance of `A`. Because objects are not "shared" nor "distributed," there is no need for any consistency protocols nor distributed garbage collection. Reference counting is required, though, in order to keep track of active bindings.

Ensuring the type-safety of incoming data is important in order to preserve the integrity of the JVM at the receiving end. In object serialization protocols such as [Jav-b], method arguments are type-checked to verify that the Java classes of the serialized data are "compatible"[2] between the caller and callee JVMs. However, in order to attain safe zero-copy transfers, primitive-valued fields (especially boolean and transient fields) and array bounds need to be checked to reinforce safety.

3.3 Versioning

One advantage of the interface-based model is that it handles implementation and interface versioning cleanly. Figure 3 illustrates a versioning example. First, the service provider decides to upgrade the implementation (because of a bug fix, for instance). As a result, the implementation of interface $\mathtt{iB_1}$ evolves from $\mathtt{B_0}$ to $\mathtt{B_1}$. Then, the provider decides to add new method calls to the service: $\mathtt{iB_1}$ is "upgraded" to $\mathtt{iB_2}$ and new implementation $\mathtt{B_2}$ is needed. Figure 3a shows the resulting Java class hierarchy and Figure 3b shows impact of the versioning process in the RPC binding. As long as the two parties use the same interface, the implementation can change without any change or recompilation having to occur at the client side. In fact, an implementation can be detached from a service and a new one (implementing the same interface) attached without clients having to rebind to the service (transition I). It is the fact that proxies are produced from the interface and not from the implementation that permits this flexibility. In addition, interfaces can be upgraded just as transparently as long as they are "upwards compatible", i.e., new interfaces extend old ones. When $\mathtt{iB_1}$ is upgraded to $\mathtt{iB_2}$ (transition II), the client need not upgrade because the proxy for $\mathtt{iB_1}$ is still compatible with the new stub for $\mathtt{iB_2}$. New proxies for $\mathtt{iB_2}$ are produced in subsequent bindings to the service, or when the client decides to "refresh" the existing binding (transition III).

3.4 Comparison with Other Models

In contrast, when a remote object changes in Java RMI [Jav-a], the clients must rebind in order to obtain an updated proxy. Java RMI has mechanisms for sending that proxy from the server's code base to the client's class loader on demand. Passing remote objects as arguments or return values of RPCs requires the proxy object itself to be marshaled and sent over the network. The advantage of the Java RMI model is that remote objects are more transparent which can be convenient when a single program is split-up into multiple parts. Unlike Java RMI, J-RPC's interface-based model is targeted more at client-server type communication than at distributed programming.

The model used in Modula-3's Network Objects [BNO+93] is similar to that of J-RPC in the sense that stubs are generated from the interface type, not from the implementation of the interface. Versioning is obtained through interface subtyping as well. The main difference is that a stub in Network Objects has dual personality: in the client side it performs RPCs to the server dispatcher, while in the server side it forwards the call to the implementation. As a result, an RPC binding is formed by a pair of stubs of the same type, whereas in J-RPC a binding can be formed by proxies and stubs of different types provided that versioning is attained through interface inheritance.

Microsoft's component object model (COM) and its distributed extension (DCOM) are similarly interface-based. COM is in essence a binary standard and components are not permitted to change their interface once they have been made public. New versions of an interface must be exported under new names and clients have to bind to the new interface in order to upgrade. One COM component can implement multiple interfaces; in particular, different versions of the same interface, such that both old and new clients can access the same component simultaneously.

---

[2] In J-RPC, class compatibility means structural equivalency, which is not necessarily the case in Java Object Serialization.

4 The Wire Protocol

The primary goal of the wire protocol is to allow zero-copy operation. That is, in most cases the object data is transmitted by the network interface DMA engine directly from the object's location and in all cases the receiving program accesses the data directly where the network interface DMA engine deposited it. For this to be feasible, the representation of an object "on the wire" must maintain the in-memory representation to the extend possible. This section describes the wire protocol used by J-RPC.

In traditional RPC systems such as SunRPC [Sun88], the presentation layer (e.g. XDR) requires data to be converted (and hence copied) at least once on both the sending and the receiving side. Highly-tuned RPC systems [JZ91, BF96] eliminate the presentation layer by generating stubs that marshal data into a format that is readily usable by the receiver but does not contain typing information. Self-describing presentation layers such as ASN.1 marshal data types into the stream but are rarely used by RPC systems due to the high decoding cost. To our knowledge, none of these protocols supports linked objects and data structures.

The wire format in J-RPC is designed with the following properties in mind:

- complete object typing information is passed to support type-checking and "portable" pointers are passed in order to support linked objects,
- fragmentation of the receive buffer is minimized by placing usable data contiguously as much as possible,
- marshaling code must be straight-forward so that it can be generated efficiently by a stub compiler,
- algorithms to type-check the objects in the receive buffer, to convert portable pointers into real pointers, and to integrate the objects into the JVM must be efficient.

4.1 Object Layout and Portable Pointers

The first design issue is to choose an object layout that is compatible with off-the-shelf JVM implementations. Given an object reference, a JVM can access the object's fields, its method table, its synchronization object, and its class descriptor. We looked at three off-the-shelf JVM implementations: Microsoft's JVM 4.79.2339 (MSVM), several versions of Sun's JDK, and Kaffe 0.10.0 beta. In MSVM and Kaffe, an object reference points to a structure containing the object fields and pointers to the method table (from which the class descriptor is accessible) and to a synchronization structure. In Sun's JDK 1.0.2, there is an additional level of indirection: an object reference points to a structure with pointers to the method table and to another structure containing the fields. For software-engineering reasons, the object layout is hidden by the native interface specification (JNI) in the latest versions of Sun JDK (1.1+). For our implementation of J-RPC we chose the layout adopted by MSVM and Kaffe for efficiency reasons — in principle, the protocol can be adjusted to Sun's JDK object layout as well.

Java objects cannot be simply sent "as-is" over the network. The fields do not contain any type information (i.e. an `int` contains just the 32-bit value) with the effect that the receiver cannot type-check them. In addition, pointers to other objects are meaningless at the receiving end. Given that all object references are also marshaled in the RPC call, these references must be converted to point to those marshaled versions.

In order to support zero-copy and group object data together in the receive buffer, J-RPC adopts a *bi-directional* layout illustrated in Figure 4a:

- The marshaled object representation starts at the *ObjRef* offset and grows to higher memory addresses while the type-descriptors of its primitive-typed fields are stored in a table (PTDT) that starts at *PTDTOff* and grows in the reverse direction. Object references are represented by positive offsets from the base address.

- The layout of a Java array (Figure 4b) contains the array type-descriptor, a flag that indicates whether it is an array of objects[3], and the length of the array followed by the elements. In case of an array of objects, the elements are just pointers to the objects. Instead of an offset to a PTDT, there is *flag* indicating whether it is an array of primitive-typed elements or an array of objects.

[3] In the case of an array of primitive-typed elements, the type of the elements in encoded in the array's type-descriptor. The flag field is used for type-checking purposes.

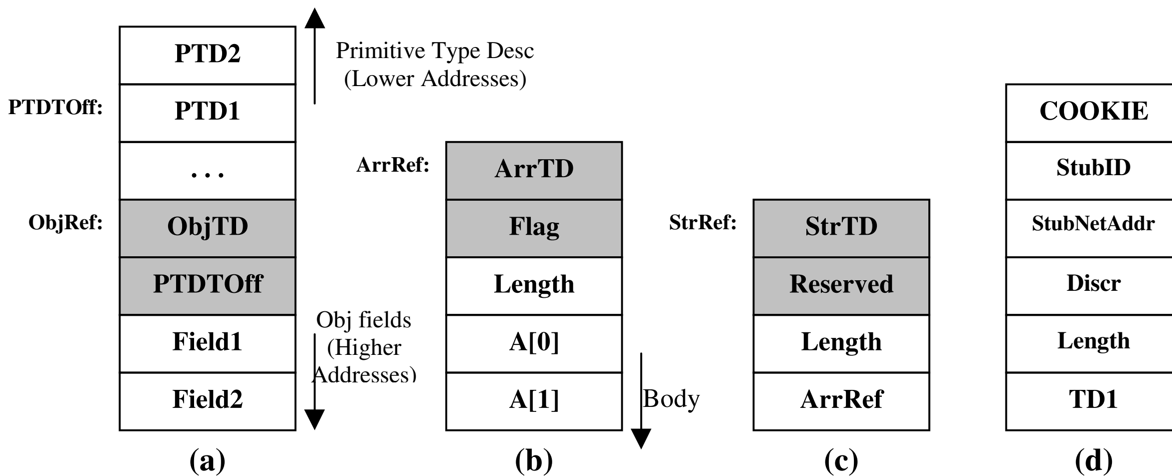


Figure 4. **(a)** Layout of a Java object. **(b)** array, **(c)** string, and **(d)** J-RPC remote interface.

- The layout of a Java string (Figure 4c) contains the string's type-descriptor, the length, and a pointer to an array of characters (strings are in unicode format).
- A remote interface is represented by a magic cookie (Figure 4d), a stub identifier, the network address where the stub is located, a network discriminator, and an ordered list of type-descriptors where the first element is the type-descriptor of the interface itself followed by the type-descriptor of its parent, and so on[4].

The process of marshaling a Java object consists of copying the object itself and all transitively-reachable ones into the network buffer in the format-described above. In order to preserve the layout of the object, marshaling of object references is delayed until the marshaling of the current object is completed. As an example, consider a Java class definition for an element of a linked list of byte arrays (`LBA`) as seen in Figure 5a. It contains an integer field `len`, a reference to a byte array `data`, and a reference to the `next` element in the list. Also, consider a remote interface `iX` that defines a method `methX` with two arguments: a boolean value, a list of byte arrays. The client imports the interface `iX`, creates a list of two elements, and calls methX. Figure 5b shows the in-memory, runtime representation of the list in Java in Microsoft's JVM, and Figure 5c shows how the list is represented in the wire using the zero-copy protocol. The base address (located at offset `OFF`) is the location of the head of the list.

4.2 Complete Protocol

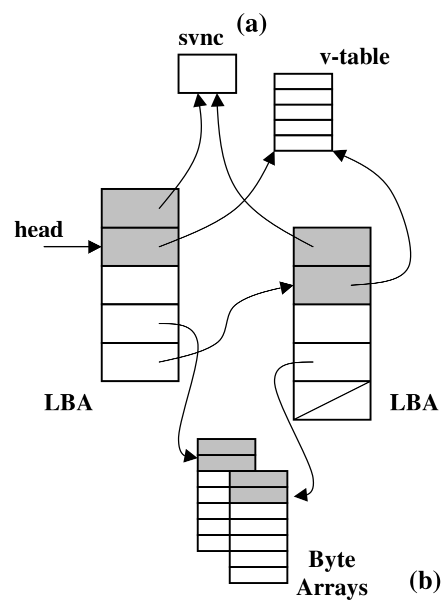
The example above shows how a Java object is represented using the bi-directional layout and portable pointers. In order to marshal all the arguments of a call, these are treated as if they were a Java class where each field corresponds to one argument. The complete protocol is depicted in Figure 5d. The base address is the location of the method identifier. The marshaling code prepends the following five additional header words: a magic cookie, the total length of the data buffer, the number of objects marshaled into the buffer, and the size of the aggregate PTDT (of all objects combined). The final buffer is referred to as an RPC *message*.

4.3 Unmarshaling

The unmarshaling procedure type-checks the incoming data, converts the portable pointers to real pointers, and integrates the object into the JVM. For efficiency, the implementation performs all three operations in one pass through the data.

[4] For complex inheritance hierarchies (i.e. interfaces with more than one parent), the list is ordered in a "breadth-first" fashion.

```
public class LBA {
  int i;
  byte[] data;
  LBA next;
}
public interface iX extends Remote{
  int methX(boolean a1, LBA a2)
    throws RemoteException;
}
public class Client {
    . . .
    iX p = RPC.Import(. . .);
    LBA head = createList(2);
    p.methX(true, head);
    . . . } }
```



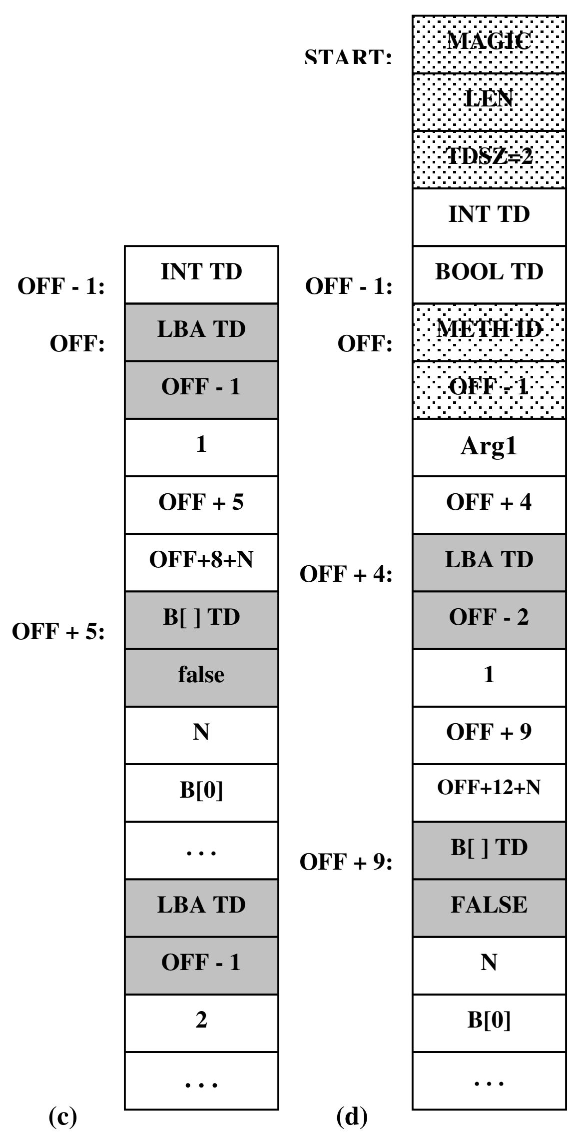


Figure 5. **(a)** Java source example, **(b)** in-memory representation of a 2-element LBA in MSVM, **(c)** corresponding J-RPC object layout for the same list, and **(d)** the complete RPC message.

An object is type-checked by first comparing its type-descriptor to the expected type-descriptor obtained at binding time. Then, the fields of the objects are type-checked using the offset to PTDT. After validating a received object, its type-descriptor is replaced by a pointer to the correct local method table, and its PTDT offset is replaced by a pointer to a synchronization structure (this is evidently JVM dependent). The complete algorithm checks and updates all non-transient objects contained in the call recursively. All offsets are bounds-checked to prevent illegal values from corrupting the receiving JVM. Type-checking an array object in addition requires validating the array-bounds.

An interface is unmarshaled by traversing the list of interface type-descriptors and finding a matching proxy from the table of registered proxies. The proxy is put into a table of active proxies indexed by the stub's JVM address. If the client terminates, active proxies are gracefully disconnected from their stubs. Interface unmarshaling may require a round-trip message to the server to increment a reference count, but techniques can be used to alleviate this overhead [EE98b, BEN+94].

4.4 Zero-Copy Transfer of Arrays

Transmitting arrays of primitive values takes advantage of the remote DMA capability in the underlying UNI to avoid copying the elements into the send buffer. The entire array is directly transferred to the receive buffer at the remote end using an RDMA. All such arrays are transferred before the send buffer, such that the arrival of the latter can initiate unmarshaling (the UNI is assumed to guarantee in-order delivery).

4.5 Summary

Careful design of the wire format is key to enabling zero-copy RPC data transfer. J-RPC incorporates typing information while maintaining the layout of the objects, supports linked objects through portable pointers, allows straight-forward marshaling and unmarshaling, and enables type-checking at the receiving end. The RPC receiver need not blindly trust the contents of the message: if a message is validated after unmarshaling, it cannot compromise the integrity of the JVM. In addition, after unmarshaling, the received objects are contiguous in memory, so fragmentation in the receive buffer occurs on a per-message instead of a per-object basis.

5 RPC System Architecture

The zero-copy protocol reduces the overheads of data transfer during an RPC to a minimum. The RPC system architecture discussed in this section (i) uses pre-allocated threads to minimize the overheads of RPC control transfer, (ii) avoids synchronization (i.e. locking) during the execution of the zero-copy protocol, and (iii) manages the receive buffer efficiently and integrates the incoming data into the JVM heap, taking garbage collection into account.

5.1 A Connection-Oriented Design: Endpoints

The RPC model based on proxies and stubs is easily supported by a connection-oriented design (Figure 6). A connection consists of a pair of endpoints, each endpoint is associated with a thread and contains thread-safe data structures for marshaling, unmarshaling, and message transmission. Connections are established between the proxy and the stub at binding time and remain open until the binding is terminated. Since each binding may have multiple connections, proxies and stubs maintain the connections in a list. The proxy synchronizes accesses to the list but no synchronization is necessary in the stub.

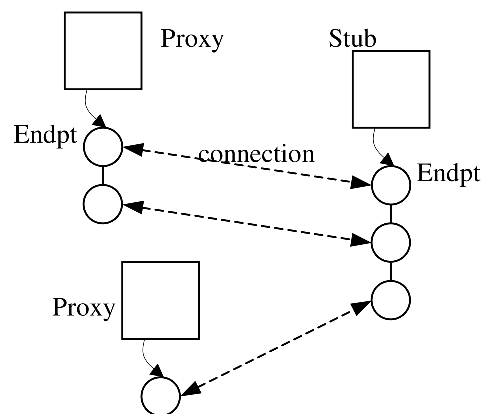


Figure 6. Connection-oriented design.

The semantics of RPC require that a new thread be available for the processing of each incoming RPC. This is due to the fact that no restrictions are placed on the operations performed in remotely invoked procedures. By associating a thread with each endpoint, the incoming RPC simply runs on that thread, eliminating the need for spawning a new thread to handle the call[5]. Also, because endpoints are thread-safe, access to most helper data structures, in particular the table mapping type descriptors to the corresponding local method table and synchronization objects, need not be synchronized[6]. The connections are specialized to handle RPC for a particular interface, so incoming RPCs can be dispatched immediately based on the method identifier. This saves the need for looking up the stub to which the call is addressed.

[5] This implementation choice obviously raises scalability issues. A more satisfactory solution would integrate the RPC needs with the thread system, but unfortunately that is difficult with current off-the-shelf JVMs.

[6] Acquiring/releasing locks are still required to look up registered proxies and stubs during interface marshaling and unmarshaling.

5.2 Mapping Endpoints onto the UNI

The J-RPC design fits into most UNI architectures because they also adopt a connection-oriented approach and provide protection on a per-connection basis. Each J-RPC endpoint is mapped to the corresponding UNI abstraction that represents a connection: a *virtual interface* in VIA.

Each endpoint has one send-area and two sets of receive-areas that are pinned in physical memory, one for regular RPC messages and another for arrays sent via RDMA. Each receive-area holds a number of messages and is gradually filled by the sender (i.e. multiple RPCs can use the same receive-area). The size of each area corresponds to the size of the largest message, which is the maximum transfer size (MTU) of the underlying NI (61Kbytes, that is, 15 pages, in the current implementation). Each endpoint also keeps one RDMA send descriptor, one receive descriptor for regular RPC messages, and a number of RDMA send descriptors for array transfers. These descriptors are located in one page of pinned memory. Each endpoint also holds tables and queues for marshaling and unmarshaling which use about 4 unpinned pages. Altogether, an endpoint currently consumes about 76 pages of pinned memory and 4 pages of unpinned memory.

The scalability of the connection-oriented design is an issue given that it consumes a significant amount of memory. This calls for a careful connection management so that idle connections are terminated in order to free resources for new connections. This issue has not been addressed by our current implementation. Commercial UNI products currently focus on clusters with small number of servers, so scalability has not been a prime concern to us.

Flow-control as well as receive-area management are performed at the sending side. During the initialization of a connection, the receiver allocates the receive-areas (in effect the receiver dictates the window size), registers them with the UNI, and transmits their addresses to the sender. From that point on, the sender decides when to transmit and where to transmit. An RPC message is sent to a new receive area if the current one is full. A notification is piggybacked with the message so the receiver can allocate a new receive-area and piggybacks its address with the reply message. This scheme incurs little overhead because the receive-area information update is fast and can often be overlapped with communication. There is no need for extra round-trip messages. Pinning and unpinning a receive-area can be expensive, but the cost is amortized over a number of messages.

The main overheads in the critical path of the sender are preparing RDMA descriptors for arrays (if any) and for the RPC message, performing flow-control checks, and posting the descriptors in the UNI send queue. When sending arrays, the RPC system ensures that the page(s) on which the arrays are located are pinned and later unpinned, which can be expensive. Updating the flow control information is fast and can be overlapped with communication. When receiving a message, the main costs are extracting the header information from the message, initializing the endpoint for method dispatch and unmarshaling, and performing flow-control duties such as preparing a new receive buffer. The latter is done infrequently and only when notified by the sender.

5.3 Memory Management

In order to support the zero-copy protocol, the receive-areas are really part of the Java heap, except that, when they are allocated, they don't contain any objects. Data is transferred into the areas by the NI and become valid objects after the unmarshaling procedure.

One concern is that the receive-areas suffer from fragmentation because the header, the PTDT, and some but not necessarily all of the Java objects in the RPC message become garbage after the RPC is completed. Currently, J-RPC delegates this problem to the Java garbage collector. If the garbage-collector is non-copying, the buffer is unpinned and returned to the Java heap. This design requires that the Java heap and the RPC system share the same free memory pool. If the GC is copying, the area can return to the endpoint's free area pool once it contains no live objects. In addition, an implementation can use Java object *finalizers* or weak pointers (provided by the Java native interface) to keep track of the number of live objects in the buffer.

6 Evaluation

We have built a prototype implementation of the RPC system using Microsoft's JVM (MSVM) and GigaNet's GNN1000 network interface. We chose MSVM over Sun's JDK because of its performance and because it provides a faster and easier to use native interface (JDirect). We rejected using freely-available implementations of JVMs for being inadequate as experimental platforms where good performance is vital — they are still immature and slow due to sub-optimal JITs.

The implementation is fully functional. It is layered directly on top of the VIA interface provided by the NI, although TCP/IP sockets can also be used to test and debug the functionality of the system. RPC proxies and stubs are written in Java manually due to the lack of an automatic stub generator and are compiled with Sun's JDK 1.1.4 compiler. The Java classes that implement the native interface used by proxies and stubs are compiled with Microsoft's Java compiler because of the JDirect support. The experiments were carried out using two 200 MHz PentiumPro running NT4.0 (SP3) with GNN1000 NI's connected point-to-point. The PCs have 128 KB of RAM, a L1 data cache of 16KB, and a L2 cache of 512KB.

One problem in the current implementation is that an off-the-shelf JVM does not allow an object allocated in native memory to be integrated into the Java heap. The garbage-collector of MSVM (a generational, copying collector) consults a list of internally allocated objects in order to implement Java object finalization correctly. A Java object that is not in that list becomes invalid after the first occurrence of garbage collection. We are currently working on this problem and as a temporary work around we actually copy the final object after it has been unmarshaled. The cost of this extra allocation and copy has been subtracted from our round-trip and bandwidth measurements.

Table 1a shows the relevant performance information of the GNN1000 NI. Table 1b shows the base performance of the MS VM in terms of method invocations, interface invocations, and Java-native crossings. All the measurements are averaged over 100,000 times.

6.1 Base-line performance

Table 2 shows the base performance of the J-RPC implementation. The round-trip latency of a null RPC is 18.4µs, 2µs higher than the raw UNI round-trip of 16.4µs. Three 1-word Java to native calls and two Java interface invocations account for about 1.2µs, and another 0.2µs due to the fact that one of the native calls is synchronized. The round-trip latency of an RPC with four word-sized arguments rises to 20.3µs due additional native calls for marshaling and unmarshaling. The round-trip latency of an RPC with one interface argument (without reference counting) is 19.7µs. An additional lock acquisition/release is needed to perform one lookup to the proxy table. This number increases as the versioning list increases[7]. Overall, these results show that the control transfer overheads are minimal for unloaded machines. The CPU utilization on both machines during the experiment was near 100%.

Figure 7 shows the latency and bandwidth numbers when transmitting arrays with RPC. We compare the performance of the one-copy and zero-copy array transfers (*ZC* and *ZCT*) in J-RPC with those of the raw VIA performance, DCOM over VIA using custom marshaling, COM over NT's LRPC, and DCOM over

Giganet GNN1000	*time (us)*
Pin Page	15.51
Unpin Page	16.44
R/T latency (C)	16.40

Java operation	*time (us)*
null method call	0.04
null interface call	0.20
1-word Java --> native call	0.25
3-word Java --> native call	0.28
acquire/release lock	0.21

Table 1. **(a)** Relevant numbers of Giganet's VIA implementation. **(b)** Base Java performance in MSVM

RPC	*R/T (us)*	*Breakdown (us)*		
		Java -> C	*Java Intf*	*Locks*
null	18.1	0.5	0.4	0.2
4-word	20.3	2.0	0.4	0.2
interface	19.7	1.0	0.4	0.4

Table 2. J-RPC base performance

[7] Due to space constraints, we've omitted a more detailed analysis of interface passing performance.

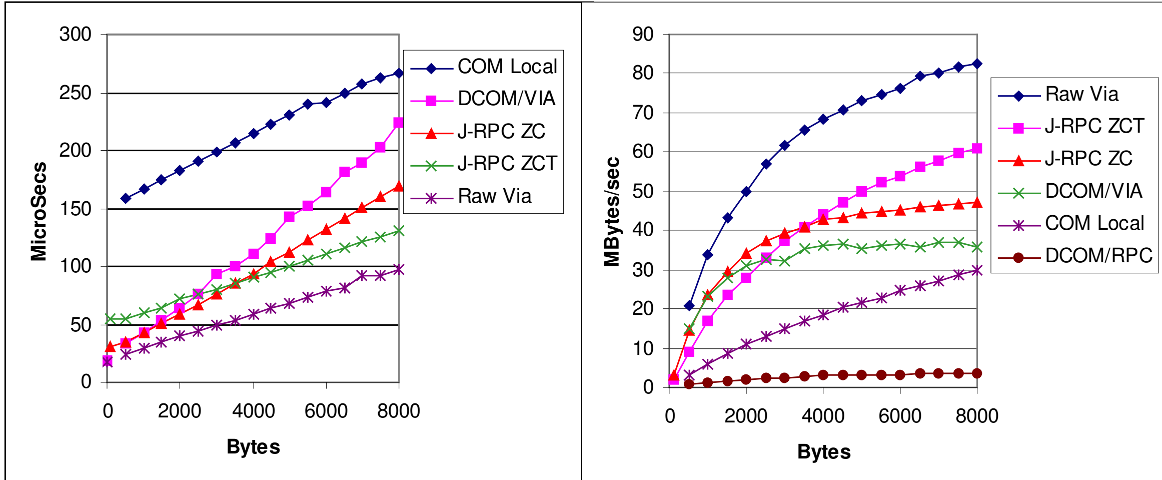


Figure 7. Latency and bandwidth for byte arrays compared to raw VIA, DCOM/VIA and COM LRPC

NT's DCE/RPC. In the *ZCT* version, the NI reads the data from the client's Java heap and performs and RDMA to the server's heap. In the *ZC* version, the Java byte array is copied into the RPC message.

The results show that the bandwidth of *ZC* is comparable to that of DCOM/VIA for array sizes smaller than 2KBytes. The bandwidth of *ZCT* is actually worse compared to *ZC* and DCOM/VIA for array sizes smaller than 2KBytes. This is attributed to the cost of pinning and unpinning memory for RDMA, which is about 15µs each. Nevertheless, the performance of *ZCT* reaches that of *ZC* at about 3.5 KBytes, and is able to achieve about 74% of the raw VIA bandwidth, compared to the 57% and 43% obtained by *ZCT* and DCOM/VIA, respectively, for sizes of around 8KBytes. It is encouraging to see that one can stream 8KB-blocks of data using Java over a fully type-safe RPC at a cost of no more than 25% of the maximum achievable bandwidth.

6.2 Zero-Copy Tradeoffs

The goal of these experiments is to evaluate the tradeoffs of zero-copy by comparing the type-safe marshaling and unmarshaling costs with direct memory-to-memory transfer as well as the costs of allocating and initializing Java objects at the receiving end. The experiment is conducted "off-wire" in order to avoid the cache misses due to DMA writes[8].

The marshaling cost (*M*) is the cost of producing a fully-typed RPC message based the J-RPC wire format. The unmarshaling cost (*U*) is the cost of type-checking, converting portable pointers, and "activating" the objects in an RPC message. Table 3 shows the *M* and *U* for several sizes of Java byte arrays. There are two kinds of *M*'s for byte arrays: one that copies the byte array into the message (*M1*) and another for zero-copy transmission (*M0*). It also shows the costs of allocating and initializing a Java byte array from native code (*AC*) and from Java, by invoking the `clone` method (*JC*). Table 3 also contains similar data for a Java linked list of 10, 100, 500 elements, where each element contains an `int` and a `next` pointer. *M*s and *U*s are listed on a per-element (*P/E*) basis.

The overall marshaling/unmarshaling cost (*M+U*) is compared to the cost of memory-copying[9] the same size of data (*M2MXfer*). *U* is compared to the costs of object allocation and initialization. We compare the size overheads of J-RPC wire format with those of Java's Object Serialization (JOS). We make the following observations from Table 4:

- For byte arrays, *M0* and *U* are constant at about 4 and 1.4 us each, and *M+U* is no worse than *M2MXfer* (ignoring cache effects) for sizes greater than 5KBytes. This shows that the bottleneck for large zero-copy data transfers is no longer the marshaling/unmarshaling code, but the memory bus.

[8] The penalty of these cache misses will be incurred whether or not zero-copy is used.

[9] *M2MXfer* of linked list is obtained by allocating a block of size N*ElementSize and issuing N mem-copies of an element's size each from the corresponding offsets of the block, where N is the number of elements in the list.

Array (bytes)	Marshal		Unmarshal		
	OneCopy (M1)	ZeroCopy (M0)	ZeroCopy (U)	Alloc&Copy (AC)	JavaClone (JC)
10	4.19	3.95	1.40	1.46	2.07
100	4.77	3.94	1.47	2.32	2.80
500	8.71	3.95	1.44	5.25	5.43
1000	12.25	3.94	1.48	8.14	8.00
5000	33.58	3.94	1.42	35.58	34.55

LL (elem)	Marshal		Unmarshal			
	Total	P/E (M)	Total	P/E (U)	Alloc&Copy P/E (AC)	JavaClone P/E (JC)
10	35.13	3.51	15.03	1.50	13.74	1.46
100	351.51	3.52	139.18	1.39	13.43	1.47
500	2257.36	4.51	699.97	1.40	13.43	1.63

Table 3. Marshaling and unmarshaling costs of byte arrays and linked lists

Array (bytes)	Comparison 1(us)			Comparison		Protocol Size		Java Obj Serial		
	M2M Xfer	M1 + U	M0 + U	1+AC/U	1+JC/U	In-Mem (bytes)	Ovhd (bytes)	Write (us)	Rea d(us)	Ovhd (bytes)
10	0.22	5.59	5.35	2.04	2.48	22	28	38	15	28
100	0.46	6.24	5.42	2.57	2.90	112	28	65	61	28
500	1.33	10.16	5.40	4.64	4.76	512	28	185	265	28
1000	1.82	13.73	5.43	6.49	6.39	1012	28	328	523	28
5000	12.07	35.00	5.36	26.04	25.31	5012	28	1492	2564	28

LL (elem)	Comp 1(P/E, us)		Comp 2(P/E, us)		Protocol Size		Java Obj Serial		
	M2M Xfer	M + U	1+AC/U	1+JC/ U	Mem (bytes)	Ovhd (bytes)	Write(us)	Read(us)	Ovhd (bytes)
10	0.45	5.02	10.14	1.10	192	32	268	226	-86
100	3.85	4.91	10.65	1.01	1632	32	3505	2234	-626
500	33.02	5.91	10.59	1.00	8032	32	20830	12020	-3026

Table 4. Analysis of zero-copy costs.

- Unmarshaling can be as much as 25x (for 5Kbyte arrays) more expensive without zero-copy at the receiving end for byte arrays. For linked lists, the unmarshaling cost can be reduced by 50% or more with zero-copy, depending on the cost of object allocation and initialization.
- The protocol adds a small constant size overhead to the raw data size. The same overhead is added by JOS for byte arrays. But for linked lists, J-RPC sends 16 bytes per element compared to JOS's 10 bytes. JOS's per-element *U* is about 17x more expensive than J-RPC's. A large fraction of this gap is due to the fact that JOS is entirely implemented in Java.

6.3 Receive-Area Behavior: RPC-based Web Server

We examine the behavior of the receive-area using traces taken from a remote debugging session of the J-Server, an extensible web server developed at Cornell [CCvE98]. The J-Server run as an extension of Microsoft's Web Server (IIS 3.0) and forwards HTTP requests using J-RPC (over sockets) to remote instances of the J-Server running on the programmer's machine. The RPC interface is a method that takes an HTTP request class and returns an HTTP response class. The HTTP request class has 20 Java strings for headers and one byte array for post data.

The traces are from a remote debugging session of a telephony application. We narrowed the traces to a stretch of about 4 minutes during which 30 HTTP requests were issued, and observed how these requests

would affect the receive-area. The sizes of each request varied from 196 bytes to 1060 bytes. The amount of live data in the receive-area at the time of each request's arrival varied from 0 to 2476 bytes, far less than the receive-area size (61 Kbytes). The fragmentation observed was minimal because an average request object was garbage-collected before the arrival of the third or fourth subsequent HTTP request.

7 Lessons Learned: VIA and off-the-shelf JVMs

Support for RDMA-writes provided by the VIA architecture has proved essential for zero-copy transmission of arrays. The sender controls whether the RDMA-write being issued consumes a receive descriptor in the receiving end, removing the need for matching send/receive descriptors. The decoupling of RDMA-writes from receives allowed us to issue a series of RDMA-writes for Java arrays to designated locations in the receive-area before sending a final RDMA-write of RPC message that wakes up the receiver.

One security concern is that a malicious sender can clobber the receive-areas by not following the flow-control protocol. VIA provides support for explicitly disabling remote reads and write. A receiver can protect itself from this threat by dedicating one receive-area per RPC invocation and asking the UNI to disable remote writes into the buffer before it is freed by the garbage collector. The drawback is that receive-areas are expensive to maintain and the majority of RPC messages are small, which leaves large areas of pinned memory unused.

The user-managed address translation for zero-copy transmission of arrays turned out to be very expensive. For small byte arrays (e.g. 500 bytes), the cost of pinning and unpinning memory makes one-way latencies 120% longer. A custom JVM design could exploit the locality of data transmissions by making the garbage collector copy live, recently transmitted byte arrays to a pinned section of the heap.

The inability to allocate or "produce" Java objects that can be directly integrated into the JVM's heap has been a major impediment. In addition, using object finalizers or weak pointers is a cumbersome way to receive notification from the garbage collector. A custom JVM can tightly integrate the RPC memory management with the Java heap: type-checked objects in RPC receive areas become valid Java objects, and the garbage collector notifies the RPC system when a receive-area can be re-used.

8 Related Work

The interface-based model is influenced by COM/DCOM, Network Objects, and Sun's Java RMI, as discussed in Section 3. Other systems such as Emerald [JLH93] and SOS [SGH+89] aim at providing full mobility of objects, while systems like Argus [LCJ+87] support reliable distributed service through using course-grained objects (i.e. processes) and atomic transactions. The main motivation behind J-RPC's model is simplicity without compromising the basic client/server computing requirements such as service revocation and versioning.

A great deal of research projects in the parallel computing such as MRPC [CcvE98], Concert [KC95], and Orca [BKT92] aimed at improving the performance of RPC on multi-computers. The main theme was to demonstrate that RPC can be efficiently layered over standard message passing libraries while reducing the overheads of method dispatch, multi-threading and synchronization. Concert depended on special compiler support for performance, while MRPC and Orca only relied on compilers for stub generation. Because multi-computers offers parallel programs dedicated access to the network fabric, security in general was not an issue.

Many high-performance RPC systems have been designed and implemented for commodity workstations. In the Firefly RPC system[SB90], data representation is negotiated at bind time and copies are avoided by direct DMA transfers from a marshaled buffer and by receive buffers statically-mapped into all address spaces. Amoeba's [TvRS+91] RPC is built on top of message passing primitives and does not enforce any specific data representation. In the Peregrine system [JZ91], arguments are passed on client stub's stack. The stub traps into the kernel so that the DMA can transfer data directly out of the stack into the server's stub stack. Peregrine also supports multi-packet RPC efficiently. The authors reported a round trip RPC overhead of 309µs on diskless Sun-3/60 connected by 10 Mbit/s Ethernet. About 50% of this overhead was due to kernel traps, context switches, and receive interrupt handling.

The RPC overheads of the above systems are dominated by kernel's involvement in the critical path. A more recent effort by the SHRIMP Fast RPC project [BF96] shows that an UNI can reduce the RPC overheads to a minimum. It achieves a round-trip latency of about 9.5µs, 1µs above the hardware minimum (between two 60 MHz Pentiums running Linux), and uses a custom format for data streams. The main motivation behind J-RPC is to investigate whether this kind of performance improvements can also be applied to a safe language system such as Java.

9 Conclusion

This paper presents a novel wire protocol for typed objects that enables zero-copy data transfer and shows how arbitrarily complex and linked objects can be marshaled and unmarshaled efficiently in a type-safe manner. This paper describes J-RPC, a software architecture for RPC in Java on top of a UNI that provides efficient system support for the wire protocol. The results show that marshaling can be done efficiently and zero-copy transmission is beneficial for large arrays. By eliminating the need for object allocation and initialization, zero-copy at the receiving end can reduce the cost of unmarshaling by factors of 1 to 25x, depending on the cost of the allocation and initialization. The result is that overall RPC performance is better than off-the-shelf RPC implementations and close to the performance of the raw NI. We have learned the shortcomings of using an off-the-shelf JVM and are currently working on integrating the J-RPC architecture into a custom JVM.

10 References

[BAL+] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. ACM Trans. on Computer Systems, 8(1), February 1990.

[BEN+94] A. Birrell, D. Evers, G. Nelson, S. Owicki, and T. Wobber. Distributed Garbage Collection for Network Objects. Systems Research Center TR, 1994.

[BF96] A. Bilas and E. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. Tech. Rep. TR-512-96, Princeton University, 1996.

[BKT92] H. Bal, M. F. Kaashoek, and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. In *IEEE Trans. on Software Engineering*, 18(3):190-205, March 1992.

[BN84] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. ACM Trans. on Computer Systems, 4(1):39-59, February 1984.

[BNO+93] Birrel, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP),* Asheville, NC, December 1993.

[CCvE98] C-C. Chang, G. Czajkowski, T. von Eicken. MRPC: A High Performance RPC System for MPMD Parallel Computing. Software—Practice and Experience (to appear)

[CMC98] Chun, B. N., A. M. Mainwaring, and D. E. Culler. *Virtual Network Transport Protocols for Myrinet*. IEEE Micro, Vol. 18, No. 1, January/February 1998.

[DBC+98] Dubnicki, C., A. Bilas, Y. Chen, S. Damianakis, and K. Li. *Shrimp Project Update: Myrinet Communication.* IEEE Micro, Vol. 18, No. 1, January/February 1998.

[EE98a] G. Eddon and H. Eddon. Inside Distributed COM. Microsoft Press, 1998.

[EE98b] G. Eddon and H. Eddon. Understanding the DCOM Wire Protocol by Analyzing Network Data Packets. Microsoft Systems Journal, March 1998, pages 45-63.

[GNN] GigaNet, Inc. http://www.giga-net.com

[Jav-a] JavaSoft. *Java RMI Specification*. http://java.sun.com.

[Jav-b] JavaSoft. *Java Object Serialization Specification*. http://java.sun.com.

[JLH93] E. Jul, H Levy, N. Hutchinso, and A. Black. Fine-grained mobility in the Emerald system. ACM Trans. on Computer Systems 6(1):109-133, February 1988.

[JZ91] D. Johnson and W. Zwaenepoel. The Peregrine high performance RPC system. Tech. Rep. COMPTR91-152, Dept. of Computer Science, Rice University, 1991.

[KC95] V. Karamcheti, and A. Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of ACM/IEEE Supercomputing*, Portland, OR, Nov 1993.

[LCJ+87] B. Liskov, D. Curtis, P. Johnson, R. Scheifler. Implementation of Argus. In Proceedings of the 11th ACM Symposium on Operating System Principles, 1987, pages 111-122.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, B. Welch. The Spring network operating system. IEEE Computer, 21(2):23-36, February 1988.

[PLC95] Pakin, S., M. Lauria, and A. Chien. *High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet*. In Proceedings of Supercomputing '95, San Diego, California, 1995.

[SB90] M. Schroeder and M. Burrows. Performance of Firefly RPC. ACM Trans. on Computer Systems, 8(1):1-17, February 1990.

[SGH+89] M. Shapiro, Y. Gourhant, S. Habert, Laurence Mosseri, M. Ruffin, C. Valot. SOS: An object-oriented operating system: assessment and perspectives. Computing Systems 2(4):287-337, Fall 1989.

[SPS98] R. Sankaran, C. Pu, and H. Shah. Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks. In Proc of USENIX NT Symposium, Seattle, WA, Aug 1998.

[Sun88] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1057, Internet

[TvRS+91] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. vanRossum. Experiences with the Amoeba distributed operating system. Comm of the ACM, 33(12):46-63, Dec 1990.

[vEBB+95] von Eicken, T., A. Basu, V. Buch, and W. Vogels. *U-Net: A User-level Network Interface for Parallel and Distributed Computing*. In Proceedings of the 15th Annual Symposium on Operating System Principles, p. 40-53, Copper Mountain Resort, Colorado, Dec. 1995.

[VIA97] The Virtual Interface Architecture. http://www.via.org