The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Lecture 4 (20T3)

Ihor Kuz, Felix Rauch, Manuel M. T. Chakravarty & Gernot Heiser

Communication

In order for processes to cooperate (e.g., work on a single task together), they must communicate. There are two reasons for this communication: synchronisation and sharing of data. Processes synchronise in order to coordinate their activities. This includes finding out whether another process is alive, determining how much of a task a process has executed, acquiring exclusive access to a resource, requesting another process to perform a certain task, etc. Processes share data about tasks that they are cooperatively working on. This may include sending data as part of a request (e.g., data to perform calculations on), returning the results of a calculation, requesting particular data, etc.

There are two ways that processes can communicate: through shared memory or through message passing. In the first case processes must have access to some form of shared memory (i.e., they must be threads, they must be processes that can share memory, or they must have access to a shared resource, such as a file). Communicating using shared memory requires processes to agree on specific regions of the shared memory that will be used to pass synchronisation information and data.

The other option (and the only option for processes that do not have access to shared memory), is for processes to communicate by sending each other messages. This generally makes use of interprocess communication (IPC) mechanisms made available by the underlying operating system. Examples of these mechanisms include pipes and sockets.

Communication in a Distributed System

While the discussion of communication between processes has, so far, explicitly assumed a uniprocessor (or multiprocessor) environment, the situation for a distributed system (i.e., a multicomputer environment) remains similar. The main difference is that in a distributed system, processes running on separate computers cannot directly access each other's memory. Nevertheless, processes in a distributed system can still communicate through either shared memory or message passing.

Message Passing

Message passing in a distributed system is similar to communication using messages in a non-distributed system. The main difference being that the only mechanism available for the passing of messages is network communication.

At its core, message passing involves two operations `send()` and `receive()`. Although these are very simple operations, there are many variations on the basic model. For example, the communication can be connectionless or connection oriented. Connection oriented communication requires that the sender and receiver first create a connection before `send()` and `receive()` can be used.

There are a number of important issues to consider when dealing with processes that communicate using message passing, which are described in the next section. Besides these variations in the message passing model, there are also issues involved with communicating between processes on heterogeneous computers. This brings up issues such as data representation and dealing with pointers, which will be discussed in more detail later.

Communication Modes

There are a number of alternative ways, or modes, in which communication can take place. It is important to know and understand these different modes, because they are used to describe the different services that a communication subsystem offers to higher layers.

A first distinction is between the two modes *data-oriented communication* and *control-oriented communication*. In the first mode, communication serves solely to exchange data between processes. Although the data might trigger an action at the receiver, there is no explicit transfer of control implied in this mode. The second mode, control-oriented communication, explicitly associates a transfer of control with every data transfer. Data-oriented communication is clearly the type of communication used in communication via shared address space and shared memory, as well as message passing. Control-oriented communication is the mode used by abstractions such as remote procedure call, remote method invocation, active messages, etc. (communication abstractions are described in the next section).

Next, communication operations can be synchronous or asynchronous. In *synchronous communication* the sender of a message blocks until the message has been received by the intended recipient. Synchronous communication is usually even stronger than this in that the sender often blocks until the receiver has processed the message and the sender has received a reply. In *asynchronous communication*, on the other hand, the sender continues execution immediately after sending a message (possibly without having received an answer).

Another possible alternative involves the buffering of communication. In the buffered case, a message will be stored if the receiver is not able to pick it up right away. In the unbuffered case the message will be lost.

Communication can also be transient and persistent. In *transient communication* a message will only be delivered if a receiver is active. If there is no active receiver process (i.e., no one interested in or able to receive messages) then an undeliverable message will simply be dropped. In *persistent communication*, however, a message will be stored in the system until it can be delivered to the intended recipient. As Figure 1 shows, all combinations of synchronous/asynchronous and transient/persistent are possible.

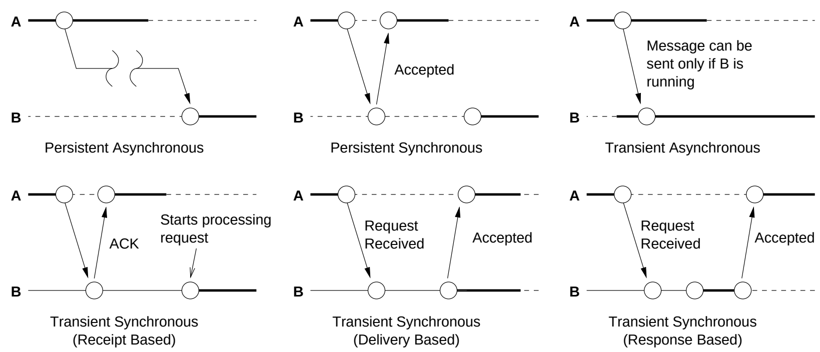

Figure 1: Possible combinations of synchronous/asynchronous and transient/persistent communication.

There are also varying degrees of reliability of the communication. With *reliable communication* errors are discovered and fixed transparently. This means that the processes can assume that a message that is sent will actually arrive at the destination (as long as the destination process is there to receive it). With *unreliable communication* messages may get lost and processes have to deal with it.

Finally it is possible to provide guarantees about the ordering of messages. Thus, for example, a communication system may guarantee that all messages are received in the same order that they are sent, while another system may make no guarantees about the order of arrival of messages.

Communication Abstractions

In the previous discussion it was assumed that all processes explicitly send and receive messages (e.g., using `send()` and `receive()`). Although this style of programming is effective and works, it is not always easy to write correct programs using explicit message passing. In this section we will discuss a number of communication abstractions that make writing distributed applications easier. In the same way that higher level programming languages make programming easier by providing abstractions above assembly language, so do communication abstractions make programming in distributed systems easier.

Some of the abstractions discussed attempt to completely hide the fact that communication is taking place. While other abstractions do not attempt to hide communication, all abstractions have in common that they hide the details of the communication taking place. For example, the programmers using any of these abstractions do not have to know what the underlying communication protocol is, nor do they have to know how to use any particular operating system communication primitives.

The abstractions discussed in the coming sections are often used as core foundations of most middleware systems. Using these abstractions, therefore, generally involves using some sort of middleware framework. This brings with it a number of the benefits of middleware, in particular the various services associated with the middleware that tend to make a distributed application programmer's life easier.

Message-Oriented Communication

The *message-oriented communication* abstraction does not attempt to hide the fact that communication is taking place. Instead its goal is to make the use of flexible message passing easier.

Message-oriented communication is based around the model of processes sending messages to each other. Underlying message-oriented communication has two orthogonal properties. Communication can be synchronous or asynchronous, and it can be transient or persistent. Whereas RPC and RMI are generally synchronous and transient, message oriented communication systems make many other options available to programmers.

Message-oriented communication is provided by *message-oriented middleware* (MOM). Besides providing many variations of the `send()` and `receive()` primitives, MOM also provides infrastructure required to support persistent communication. The `send()` and `receive()` primitives offered by MOM also abstract from the underlying operating system or hardware primitives. As such, MOM allows programmers to use message passing without having to be aware of what platforms their software will run on, and what services those platforms provide. As part of this abstraction MOM also provides marshalling services. Furthermore, as with most middleware, MOM also provides other services that make building distributed applications easier.

MPI (Message Passing Interface) is an example of a MOM that is geared toward high-performance transient message passing. MPI is a message passing library that was designed for parallel computing. It makes use of available networking protocols, and provides a huge array of functions that basically perform synchronous and asynchronous `send()` and `receive()`.

Another example of MOM is MQ Series from IBM. This is an example of a *message queuing system*. Its main characteristic is that it provides persistent communication. In a message queuing system, messages are sent to other processes by placing them in queues. The queues hold messages until an intended receiver extracts them from the queue and processes them. Communication in a message queuing system is largely asynchronous.

The basic queue interface is very simple. There is a primitive to append a message onto the end of a specified queue, and a primitive to remove the message at the head of a specific queue. These can be blocking or nonblocking. All messages contain the name or address of a destination queue.

Messages can only be added to and retrieved from local queues. Senders place messages in *source queues* (or *send queues*), while receivers retrieve messages from *destination queues* (or *receive queues*). The underlying system is responsible for transferring messages from source queues to destination queues. This can be done simply by fetching messages from source queues and directly sending them to machines responsible for the appropriate destination queues. Or it can be more complicated and involve relaying messages to their destination queues through an overlay network of routers. An example of such a system is shown in Figure 2. In the figure, an application on sender A sends a message to an application on receiver B. It places the message in its local source queue, from where it is forwarded through routers R1 and R2 into the receiver's destination queue.

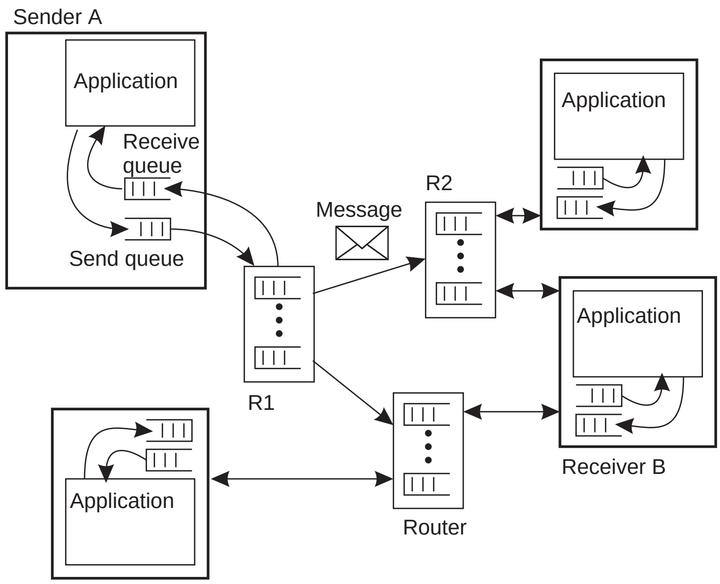

Figure 2: An example of a message queuing system.

Remote Procedure Call (RPC)

The idea behind a *remote procedure call* (RPC) is to replace the explicit message passing model with the model of executing a procedure call on a remote node [BN84]. A programmer using RPC simply performs a procedure call, while behind the scenes messages are transferred between the client and server machines. In theory the programmer is unaware of any communication taking place.

Figure 3 shows the steps taken when an RPC is invoked. The numbers in the figure correspond to the following steps (steps seven to eleven are not shown in the figure):

1. client program calls client stub routine (normal procedure call)
2. client stub packs parameters into message data structure (marshalling)
3. client stub performs `send()` syscall and blocks
4. kernel transfers message to remote kernel
5. remote kernel delivers to server stub procedure, blocked in `receive()`
6. server stub unpacks message, calls service procedure (normal procedure call)
7. service procedure returns to stub, which packs result into message

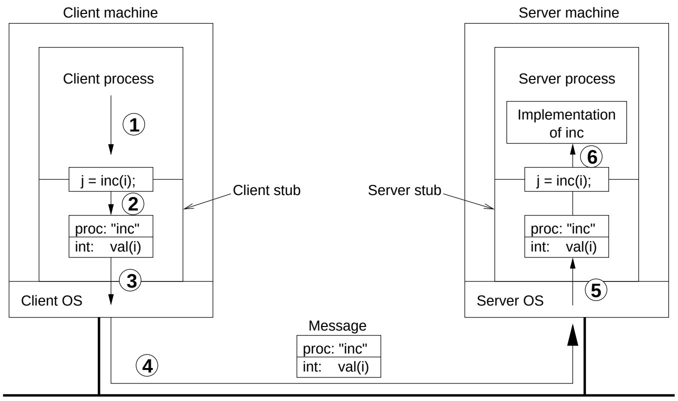

Figure 3: A remote procedure call.

8. server stub performs `send()` syscall
9. kernel delivers to client stub
10. client stub unpacks result (unmarshalling)
11. client stub returns to client program (normal return from procedure)

A server that provides remote procedure call services defines the available procedures in a *service interface*. A service interface is generally defined in an *interface definition language* (IDL), which is a simplified programming language, sufficient for defining data types and procedure signatures but not for writing executable code. The IDL service interface definition is used to generate client and server stub code. The stub code is then compiled and linked in with the client program and service procedure implementations respectively.

The first widely used RPC framework was proposed by Sun Microsystems in Internet RFC1050 (currently defined in RFC1831). It is based on the XDR (External Data Representation) format defined in Internet RFC1014 (currently defined in RFC4506) and is still being heavily used as the basis for standard services originating from Sun such as NFS (Network File System) and NIS (Network Information Service). Another popular RPC framework is DCE (Distributed Computing Environment) RPC, which has been adopted in Microsoft's base system for distributed computing.

More modern RPC frameworks are based on XML as a data format and are defined to operate on top of widely used standard network protocols such as HTTP. This simplifies integration with Web servers and is useful when transparent operation through firewalls is desired. Examples of such frameworks are XML-RPC and the more powerful, but often unnecessarily complex SOAP.

As mentioned earlier there are issues involved with communicating between processes on heterogeneous architectures. These include different representations of data, different byte orderings, and problems with transferring pointers or pointer-based data structures. One of the tasks that RPC frameworks hide from programmers is the packing of data into messages (*marshalling*) and unpacking data from messages (*unmarshalling*). Marshalling and unmarshalling are performed in the stubs by code generated automatically from IDL compilers and stub generators.

An important part of marshalling is converting data into a format that can be understood by the receiver. Generally, differences in format can be handled by defining a standard *network format* into which all data is converted. However, this may be wasteful if two communicating machines use the same internal format, but that format differs from the network format. To avoid this problem, an alternative is to indicate the format used in the transmitted message and rely on the receiver to apply conversion where required.

Because pointers cannot be shared between remote processes (i.e., addresses cannot be transferred verbatim since they are usually meaningless in another address space) it is necessary to flatten, or serialise, all pointer-based data structures when they are passed to the RPC client stub. At the server stub, these serialised data structures must be unpacked and recreated in the recipient's address space. Unfortunately this approach presents problems with aliasing and cyclic structures. Another approach to dealing with pointers involves the server sending a request for the referenced data to the client every time a pointer is encountered.

In general the RPC abstraction assumes synchronous, or blocking, communication. This means that clients invoking RPCs are blocked until the procedure has been executed remotely and a reply returned. Although this is often the desired behaviour, sometimes the waiting is not necessary. For example, if the procedure does not return any values, it is not necessary to wait for a reply. In this case it is better for the RPC to return as soon as the server acknowledges receipt of the message. This is called an *asynchronous RPC*.

It is also possible that a client does require a reply, but does not need it right away and does not want to block for it either. An example of this is a client that prefetches network addresses of hosts that it expects to contact later. The information is important to the client, but since it is not needed right away the client does not want to wait. In this case it is best if the server performs an asynchronous call to the client when the results are available. This is known as *deferred synchronous RPC*.

A final issue that has been silently ignored so far is how a client stub knows where to send the RPC message. In a regular procedure call the address of the procedure is determined at compile time, and the call is then made directly. In RPC this information is acquired from a *binding service*; a service that allows registration and lookup of services. A binding service typically provides an interface similar to the following:

- `register(name, version, handle, UID)`
- `deregister(name, version, UID)`
- `lookup(name, version)` → `(handle, UID)`

Here `handle` is some physical address (IP address, process ID, etc.) and `UID` is used to distinguish between servers offering the same service. Moreover, it is important to include version information since the flexibility requirement for distributed system requires us to deal with different versions of the same software in a heterogeneous environment.

Remote Method Invocation (RMI)

When using RPC, programmers must explicitly specify the server on which they want to perform the call (possibly using information retrieved from a binding service). Furthermore, it is complicated for a server to keep track of the different state belonging to different clients and their invocations. These problems with RPC lead to the *remote method invocation* (RMI) abstraction. The transition from RPC to RMI is, at its core, a transition from the server metaphor to the object metaphor.

When using RMI, programmers invoke methods on remote objects. The object metaphor associates all operations with the data that they operate on, meaning that state is encapsulated in the remote object and much easier to keep track of. Furthermore, the concept of remote object, improves location transparency: once a client is bound to a remote object, it no longer has to worry about where that object is located. Also, objects are first-class citizens in an object-based model, meaning that they can be passed as arguments or received as results in RMI. This helps to relieve many of the problems associated with passing pointers in RPC.

Although, technically, RMI is a small evolutionary step from RPC, the model of remote and distributed objects is very powerful. As such, RMI and distributed objects form the base for a widely used distributed systems paradigm, and will be discussed in detail in a future lecture.

The Danger of Transparency

Unfortunately, the illusion of a procedure call is not perfect for RPCs and that of a method invocation is not perfect for RMI. The reason for this is that an RPC or RMI can fail in ways that a "real" procedure call or method invocation cannot. This is due to the problems such as not being able to locate a service (e.g., it may be down or have the wrong version), messages getting lost, servers crashing while executing a procedure, etc. As a result, the client code has to handle error cases that are specific to RPCs.

In addition to the new failure modes, the use of threads (to alleviate the problem of blocking) can lead to problems when accessing global program variables (like the POSIX `errno`). Moreover, some forms of arguments like varargs in C do not lend themselves well to the static generation of marshalling code. As mentioned earlier, pointer-based structures also require extra attention, and exceptions, such as user interrupts via keyboard, are more difficult to handle.

Furthermore, RPC and RMI involve many more software layers than local system calls and also incur network latencies. Both form potential performance bottlenecks. The code must, therefore, be carefully optimised and should use lightweight network protocols. Moreover, since copying often dominates the overhead, hardware support can help. This includes DMA directly to/from user buffers and scatter-gather network interfaces that can compose a message from data at different addresses on the fly. Finally, issues of concurrency control can show up in subtle ways that, again, break the illusion of executing a local operation. These problems are discussed in detail by Waldo et al. [WWWK94].

Group Communication

Group communication provides a departure from the point-to-point style of communication (i.e., where each process communicates with exactly one other process) assumed so far. In this model of communication a process can send a single message to a group of other processes. Group communication is often referred to as *broadcast* (when a single message is sent out to everyone) and *multicast* (when a single message is sent out to a predefined group of recipients).

Group communication can be applied in any of the previously discussed system architectures. It is often used to send requests to a group of replicas, or to send updates to a group of servers containing the same data. It is also used for service discovery (e.g., broadcast a request saying "who offers this service?") as well as event notification (e.g., to tell everyone that the printer is on fire).

Issues involved with implementing and using group communication are similar to those involved with regular point-to-point communication. This includes reliability and ordering. The issues are made more complicated because now there are multiple recipients of a message and different combinations of problems may occur. For example, what if only one of the recipients does not receive a message, should it be multicast out to everyone again, or only to the process that did not receive it? Or, what if messages arrive in a different order on the different recipients and the order of messages is important?

A widely implemented (but not as widely used) example of group communication is IP multicast. The IP multicast specification has existed for a long time, but it has taken a while for implementations to make their way into the routers and gateways (which is necessary for it to become viable).

An increasingly important form of group communication is *gossip-based communication*, which is often used for data dissemination. This technique relies on epidemic behaviour like diseases spreading among people. One variant is *rumour spreading* (or simply *gossiping*), which resembles the way in which rumours spread in a group of people. In this form of communication, a node A that receives some new information will contact an arbitrary other node B in the system to push the data to that node. If node B did not have the data previously, nodes A and B will continue to contact other nodes and push the data to them. If, however, node B already had that data, then node A stops spreading the data with a certain probability. Gossiping cannot make any guarantees that all nodes will receive all data, but works quite well in practice to disseminate data quickly.

Event-Based Communication

The event-based abstraction decouples senders and receivers in communication. Senders produce events (which may carry data), without specifying a receiver for them. Receivers listen for events that they are interested in, without specifying specific senders. The underlying middleware is responsible for delivering appropriate events to appropriate receivers. A common category of event-based communication are *publish/subscribe* systems, where senders publish events, and receivers subscribe to events of interest. Subscriptions can be based on topic, content, or a more complex combination of event and context properties.

Distributed Shared Memory

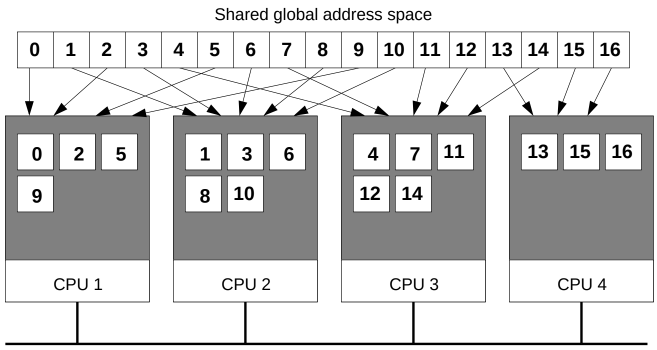

Figure 4: Distributed Shared Memory: pages in a global address space distributed over four independent computers.

Because distributed processes cannot access each other's memory directly, using shared memory in a distributed system requires special mechanisms that emulate the presence of directly accessible shared memory. This is called *distributed shared memory* (DSM). The idea behind DSM is that processes on separate computers all have access to the same virtual address space. The memory pages that make up this address space actually reside on separate computers. Whenever a process on one of the computers needs to access a particular page, it must find the computer that actually hosts that page and request the data from it. Figure 4 shows an example of how a virtual address space might be distributed over various computers.

There are many issues involved in the use and design of distributed shared memory. As such, a separate lecture will be dedicated to a detailed discussion of DSM.

Tuple Spaces

A *tuple space* is an abstraction of distributed shared memory into a generalised shared space. In this model of communication, processes place tuples containing data into the space, while others can search the space, and read and remove tuples from the space. The underlying middleware coordinates the placement and removal of the tuples, and ensures that properties such as ordering and consistency are maintained.

Streams

Whereas the previous communication abstractions dealt with discrete communication (that is they communicated chunks of data), the Stream abstraction deals with continuous communication, and in particular with the sending and receiving of *continuous media*. In continuous media, data is represented as a single stream of data rather than discrete chunks (for example, an email is a discrete chunk of data, a live radio program is not). The main characteristic of continuous media is that besides a spatial relationship (i.e., the ordering of the data), there is also a temporal relationship between the data. Film is a good example of continuous media. Not only must the frames of a film be played in the right order, they must also be played at the right time, otherwise the result will be incorrect.

A stream is a communication channel that is meant for transferring continuous media. Streams can be set up between two communicating processes, or possibly directly between two devices (e.g., a camera and a TV). Streams of continuous media are examples of *isochronous communication*, that is communication that has minimum and maximum end-to-end time delay requirements.

When dealing with isochronous communication, *quality of service* is an important issue. In this case quality of service is related to the time dependent requirements of the communication. These requirements describe what is required of the underlying distributed system so that the temporal relationships in a stream can be preserved. This generally involves timeliness and reliability.

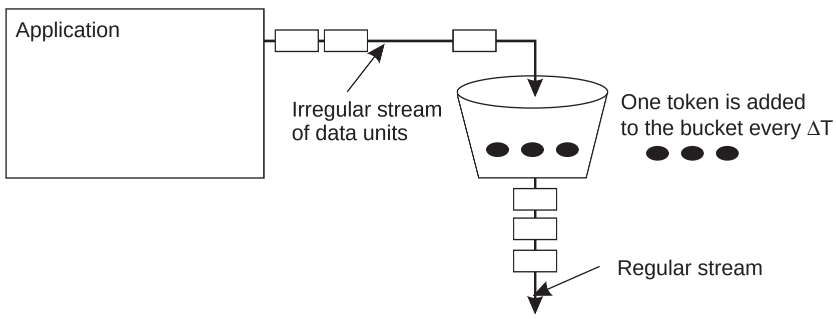

Figure 5: The token bucket model.

Quality of service requirements are often specified in terms of the parameters of a token bucket model (shown in Figure 5). In this model tokens (permission to send a fixed number of bytes) are regularly generated and stored in a bucket. An application wanting to send data removes the required amount of tokens from the bucket and then sends the data. If the bucket is empty the application must wait until more tokens are available. If the bucket is full newly generated tokens are discarded.

It is often necessary to synchronise two or more separate streams. For example, when sending stereo audio it is necessary to synchronise the left and right channels. Likewise when streaming video it is necessary to synchronise the audio with the video.

Formally, synchronisation involves maintaining temporal relationships between substreams. There are two basic approaches to synchronisation. The first is the client based approach, where it is up to the client receiving the substreams to synchronise them. The client uses a synchronisation profile that details how the streams should be synchronised. One possibility is to base the synchronisation on timestamps that are sent along with the stream. A problem with client side synchronisation is that, if the substreams come in as separate streams, the individual streams may encounter different communication delays. If the difference in delays is significant the client may be unable to synchronise the streams. The other approach is for the server to synchronise the streams. By multiplexing the substreams into a single data stream, the client simply has to demultiplex them and perform some rudimentary synchronisation.

References

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., 1994.