22. Communication Schemes

In this chapter, we build on top of the synchronization primitives explored so far, and introduce three common classes of multi-party communication schemes: Remote-Procedure Call (RPC), Message Queues (MQ) and Publish/Subscribe (P/S).

22.1 Inter-Process Communication

As software and platforms become increasingly more complex, the requirement for parallelization becomes more and more stringent. Depending on the problem at hand, parallelization can significantly speed-up computation, in a way that is linear with the number of available processors. This is the idea behind GPUs: instead of having few, very complex full-fledged CPUs, GPUs use a different approach: they define hundreds, or thousands of simple, specialized computing cores. These cores process perform the same exact processing logic, just on different data (same control plane, different data planes). This is also known as Same-Instruction/Multiple-Data (SIMD) computation. The point is: there are problems that can be efficiently solved via parallilization. In general, problems in vision, heavy data processing, learning, and really anything that can be reduced to a matrix multiplication problem can be efficiently parallelized.

A common denominator in any parallel algorithm is the following: multiple parties (agents) need do co-operate to reach a common goal. Since the goal is common, the execution of the agents cannot be completely independent. In other words: agents will need to **communicate** more or less heavily with each other.

So far, we have explored the problem of having multiple agents synchronize over one or more shared resources. We have demonstrated that in this way it is possible to perform execution rendezvous, as well as mutual exclusion. In this sense, it is clear that communication among multiple parties is achievable only if appropriate synchronization can be performed. Simply put,

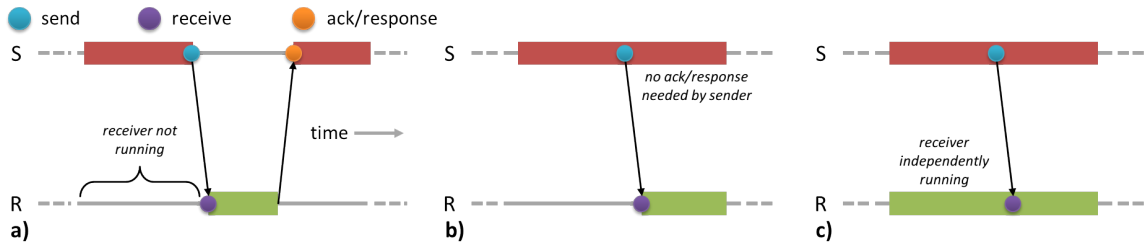

Figure 22.1: High-level diagram of (a) fully synchronous, (b) semi-asynchronous, and (c) fully asynchronous communication between a sender (S) and a receiver (R).

synchronization can be considered as a building block for communication. Strict synchronization using semaphores and spinlocks, and loose synchronization using RCUs has been described. In all the cases considered so far, we focused on a system where multiple agents can access the same memory space. This is generally true in a multi-processor platform.

Clearly, different synchronization schemes are required if the assumption of a single memory space is not satisfied. For instance: how to synchronize agents on different systems in a network? Often, this is problem is solved by introducing an intermediate software layer[1], namely a **middleware**. A middleware, abstracts away the fact that agents are physically located on different nodes, and provides synchronization primitives for systems where agents can be co-located or **distributed** on a network.

Thus, let us assume that a middleware is available. We can now focus on the possible communication schemes implemented on top of the middleware, without worrying about the nitty gritty details of the implementation. Without even worrying about the physical location of the agents. Without loss of generality, any communication scheme involves a set of sender agents, and receiver agents. Clearly, the role of different agents can change overtime. Depending on the behavior of the agents as they send/receive data, communication can be synchronous or asynchronous. Depending on the persistence of data dependently or independently from the status of senders and receivers, communication can be transient or persistent. Let us now review these cases.

22.1.1 Synchronous vs. Asynchronous Communication

If the communication is **fully synchronous**, both the send and receive operations are blocking. In this case, the sender will block waiting for a receiver to complete processing on the submitted piece of information. At the same time, the receiver will not perform any computation unless prompted by a sender. Figure 22.1a depicts the case of fully synchronous communication between a sender and a receiver.

On the other hand, if the result of receiver-side computation is not required by the sender to continue, then the sender may submit a piece of information and continue. In this case, if the job of the receiver is uniquely to process sender-originated inputs, or if further computation on the receiver side cannot be performed without new inputs, the receiver will still block on receive. This

[1] After all, computer science is the science of creating new abstractions to hide complexity.

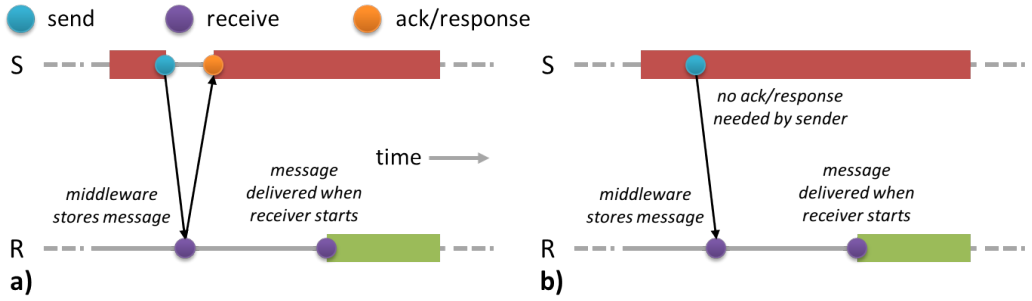

Figure 22.2: High-level diagram of persistent communication scheme. Depending on the behavior of the sender, we can have (a) persistent & synchronous, or (b) persistent & asynchronous communication between a sender (S) and a receiver (R).

communication scheme is **asynchronous** for senders and **synchronous** for receivers. We call this a case of **semi-asynchronous** communication, depicted in Figure 22.1b.

Finally, if both send and receive are non-blocking, we are in the presence of a **fully asynchronous** communication scheme. In this case, not only the sender can carry on with its own business after having sent a message, but the receiver will also be able to continue its logic even if no new message has been received. This case is depicted in Figure 22.1c.

22.1.2 Persistent vs. Transient Communication

Next, let us consider what happens to the messages as they are sent/received. We can have two cases.

First, assume that the sender sends its message and terminates. Or that at the time the message is sent, the receiver is not in execution yet, or its is unable to receive the message. If any of these cases occur and the message is lost, then we are in the presence of **transient** communication. This is the typical case for transport-level network communication: a network router will attempt to forward a certain message, but will drop it if no suitable route is found (or in case of congestion).

Conversely, if the message remains in the middleware in spite of sender's termination and/or receiver's inability to process the message, we say that the middleware implements a **persistent** communication scheme. Depending on the behavior of the senders and receivers, persistent communication can also be synchronous or asynchronous.

Consider persistent communication. In the former case, depicted in Figure 22.2a, the sender will block waiting for an acknowledgment from the middleware that the message has been stored. In the latter case, no such reception acknowledgment is required by the sender and the sender may continue unblocked, as shown in Figure 22.2b.

In case of transient communication, Figure 22.3 depicts what happens in case of synchronous communication. In the first case, covered in Figure 22.3a, the receiver is ready to accept an incoming message. Hence, the message will be delivered and the sender will block until receiver's response. If the receiver is not ready to handle a request (e.g. it is not running or busy), then communication fails

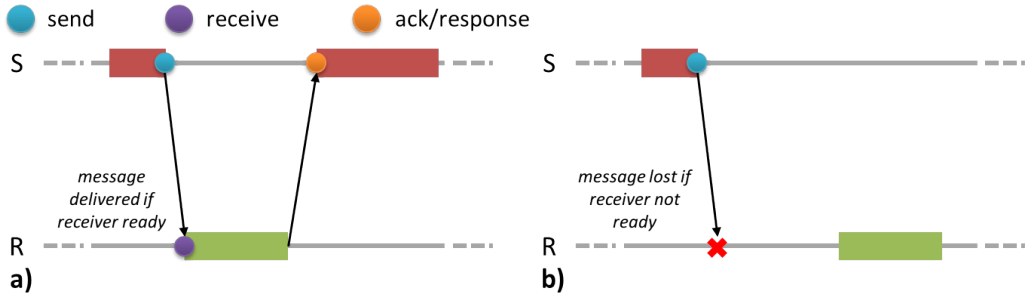

Figure 22.3: High-level diagram of transient & synchronous communication scheme. Depending on the readiness of the receiver, we can have that (a) communication is successful, or (b) a communication error occurs between a sender (S) and a receiver (R).

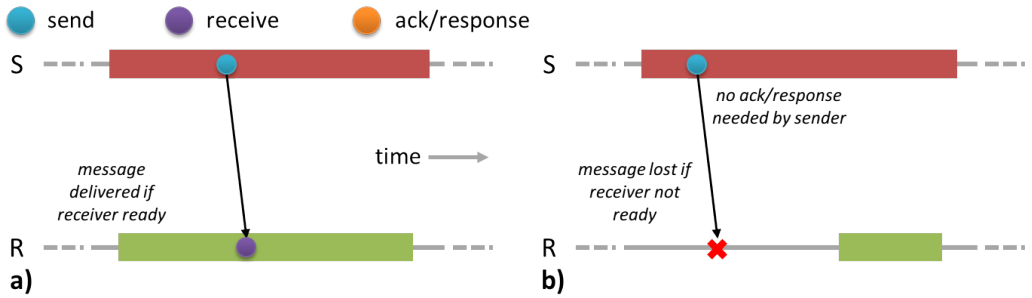

Figure 22.4: High-level diagram of transient & asynchronous communication scheme. Depending on the readiness of the receiver, we can have that (a) communication is successful, or (b) a communication error occurs between a sender (S) and a receiver (R).

with an error, as shown in Figure 22.3b. This may lead to a termination of the sender, as shown in the figure. Alternatively, the sender may recover from the error and resume its execution.

Finally, the case of transient and asynchronous communication is depicted in Figure 22.4. Once again, the receiver could be ready to handle the incoming message/request in which case successful communication is achieved, as shown in Figure 22.4a. Conversely, the sent message is lost if the receiver is not ready or busy. This case is captured by Figure 22.4b. Note that in this case the sender continues its execution without being affected by the communication error.

Table 22.1 summarizes the different combinations of synchronous, asynchronous, persistent, transient communication schemes.

22.2 Remote Procedure Call – RPC

Have you ever written a piece of code? Probably yes. Have you defined functions in your code? I hope that the answer is still yes. In this case, you know how a function call (or procedure call) works. You first pass some parameters, then invoke the considered function. In this case, the caller will wait

	Persistent	**Transient**
Synchronous	(a) Guaranteed message delivery, sender blocked	(c) Sender blocked until receiver's response, or delivery failure (e.g. RPC)
Asynchronous	(b) Sender not blocked, message stored (e.g. e-mails)	(d) Sender does not wait, undeliverable messages lost (e.g. UDP)

Table 22.1: Possible combinations of communication schemes.

until the callee has completed, then use the obtained result.

In case of **remote procedure call**, the same logic is followed. The callee, however, could correspond to an agent physically located somewhere else in the (distributed) system. In a distributed system, it may happen that the callee is not available to handle the call. In this case, a possible outcome of the call is an error. The call fails and the sender is unblocked.

Because the semantics of RPC impose that both sender and receiver need to block, RPC is an example of a **synchronous** communication scheme. In RPC, moreover, the callee needs to be ready to handle the response. In case this is not true, the RPC fails and the passed parameters are discarded by the middleware. As such, RPC is also a **transient** scheme. RPC is a widely spread communication scheme that is supported even without the need of a dedicated middleware in many modern programming languages. Some examples? Here you go:

- Java Remote Method Invocation (RMI);
- Distributed Ruby;
- Remote Python Call (RPyC);
- JSON Remote Procedure Call (JSON-RPC);
- BERT-RPC: a cross-language RPC protocol. This is extensively used by GitHub[2].

Note that there exist extensions of RPC that allow the sender to continue its execution without waiting for the receiver's response. This variant is also called asynchronous RPC. Asynchronous RPC can be implemented if the sender does not require the response of the receiver to continue (e.g. `void` methods); or in asynchronous programming languages, like Node.JS.

22.3 Message Queues – MQ

In the context of synchronization, we have discussed how to implement a multi-party producer consumer communication scheme. As it turns out, this scheme is also valid in distributed systems. In a traditional multi-party producer/consumer approach there exist a shared buffer of produced objects ready to be consumed. In its generalized formulation that works beyond the single platform, the middleware defines one or more **queues of messages**. For this reason, the scheme goes under the name of Message Queue (MQ). The corresponding middleware is often called Message Oriented Middleware (MOM).

[2] `https://github.com/blog/530-how-we-made-github-fast`

The typical **sender behavior** in a MQ is **asynchronous**. The sender submits ("put" operation) the message to the queue and goes on with its own execution. Queues in MQs are of virtually infinite size, so the case in which the queue is full is not likely. In the unlikely event of a full queue, the sender does not block. Rather, the send operation fails with an error. At the same time, a receiver who finds an empty queue typically blocks on a "get" operation. Message queues, hence, are **synchronous from the receiver's side**. It follows that this is a **semi-asynchronous** communication scheme.

Since sent messages are inserted in a middleware-maintained queue, messages persist even if the sender terminates, or the receiver is not ready. For this reason, MQs represent a **persistent** communication schemes. So what technologies are available out there that implement MQs? Here are few of them:

- IBM MQ;
- Microsoft Message Queuing (MSMQ);
- Apache ActiveMQ;
- Amazon Simple Queue Service (SQS);
- Open Message Queue (OpenMQ);
- RabbitMQ;

Keep in mind that once you go through the trouble of implementing a shared message queue for a distributed system, you may as well provide additional operations aside from the "put" and "get" operations mentioned above. It turns out that once a MQ provides additional operations that change the behavior of senders and receivers, we are in the presence of a new communication scheme. See next section.

22.4 Publish / Subscribe

Think about how emails work. You declare that you are interested in all the messages sent to `your.name@bu.edu`. But you keep carrying on with your day even if you do not receive any message in your inbox. On the other hand, whoever want to reach you can write an email to `your.name@bu.edu`. However, he/she will not be waiting for you to receive/read/respond to the message. He/she will be carrying on with his/her day instead.

This communication approach describes a scheme that is **fully asynchronous**, i.e. asynchronous for both the reader and the receiver. More in general, the scheme goes under the name of **Publish / Subscribe**. In P/S, there exists a list of *topics* where publishers (i.e. senders) can submit new messages. On the other hand, subscribers (i.e. receivers) declare to the middleware that they are interested in a given set of topics. They also provide to the middleware a function that the middleware will call whenever new data for any of the subscribed topics is available. This function is called a **notify callback** function.

Hence, we have three main operations in this scheme: (i) "publish", invoked by a publisher when it produces new data; (ii) "subscribe" invoked by a subscriber to declare interest in a topic; and (iii) "notify" invoked by the middleware on the subscribers when new data is available for topics to which

they are interested.

The typical implementation of a P/S scheme keeps a queue of messages for each topic. The message can however be deleted if all the subscriber have read the message at least once. This approach is clearly **persistent** and goes under the name of event-based publish/subscribe. In some cases, it may be useless to keep older messages. Think about a system with sensors and logic using sensor values. If a more recent sensor reading is available, there may be little or no reason to keep older readings. If new subscriber-generated message overwrite older messages, the communication scheme is **transient**. The latter scheme goes under the name of state-based publish/subscribe.

Apache Kafka is a popular publish/subscribe engine that was originally developed by LinkedIn in 2010 and became an open-source project in 2011. Similarly, the Robot Operating System (ROS) used in many complex robotic applications to handle sensor fusion and control, uses a publish/subscribe communication scheme between tasks that perform sensing, control, and actuation.