1.5 Primitives for distributed communication

Blocking/non-blocking, synchronous/asynchronous primitives

Message send and message receive communication primitives are denoted Send() and Receive(), respectively.

A Send primitive has at least two parameters – the destination, and the buffer in the user space, containing the data to be sent. Similarly, a Receive primitive has at least two parameters – the source from which the data is to be received (this could be a wildcard), and the user buffer into which the data is to be received.

There are two ways of sending data when the Send primitive is invoked – the buffered option and the unbuffered option.

The buffered option which is the standard option copies the data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network. In the unbuffered option, the data gets copied directly from the user buffer onto the network.

For the Receive primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.

The following are some definitions of blocking/non-blocking and synchronous/ asynchronous primitives.

- **Synchronous primitives**: A Send or a Receive primitive is synchronous if both the Send() and Receive() handshake with each other. The processing for the Send primitive completes only after the invoking processor learns that the other corresponding Receive primitive has also been invoked and that the receive operation has been completed. The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.
- **Asynchronous primitives:** A Send primitive is said to be asynchronous if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer. It does not make sense to define asynchronous Receive primitives.

- **Blocking primitives:** A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.
- **Non-blocking primitives**: A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed. For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer. For a non-blocking Receive, control returns to the process even before the data may have arrived from the sender.

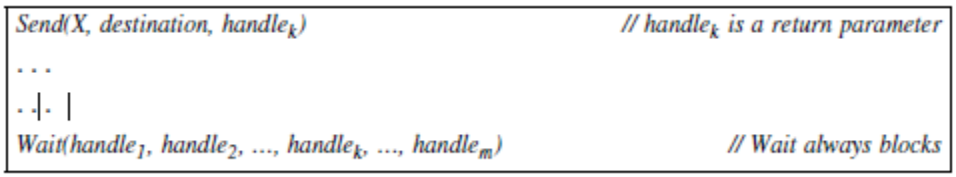

Fig. A non-blocking send primitive.

The code for a non-blocking Send would look as shown in Figure. First, it can keep checking (in a loop or periodically) if the handle has been flagged or posted. Second, it can issue a Wait with a list of handles as parameters. The Wait call usually blocks until one of the parameter handles is posted.

If at the time that Wait() is issued, the processing for the primitive (whether synchronous or asynchronous) has completed, the Wait returns immediately. The completion of the processing of the primitive is detectable by checking the value of $handle_k$. If the processing of the primitive has not completed, the Wait blocks and waits for a signal to wake it up. When the processing for the primitive completes, the communication subsystem software sets the value of $handle_k$ and wakes up (signals) any process with a Wait call blocked on this $handle_k$ This is called posting the completion of the operation.

There are therefore four versions of the Send primitive – synchronous blocking, synchronous non-blocking, asynchronous blocking, and asynchronous non-blocking. For the Receive primitive, there are the blocking synchronous and non-blocking synchronous versions. These versions of the primitives are illustrated in Figure using a timing diagram. Here the timelines

are shown for each process: (1) for the process execution, (2) for the user buffer from/to which data is sent/received, and (3) for the kernel/communication subsystem.

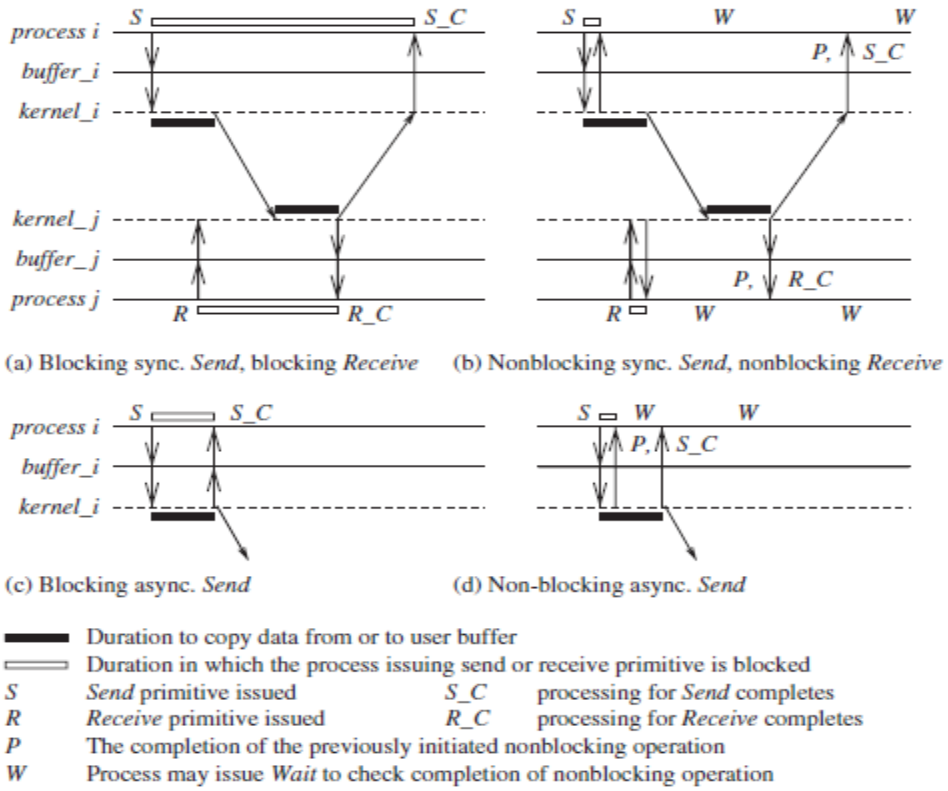

Figure: Blocking/ non-blocking and synchronous/asynchronous primitives.
Process Pi is sending and process Pj is receiving.
(a) Blocking synchronous Send and blockin (synchronous) Receive.
(b) Non-blocking synchronous Send and nonblocking (synchronous) Receive.
(c) Blocking asynchronous Send.
(d) Non-blocking asynchronous Send.

Processor synchrony

Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized. As this synchrony is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a step, the processors are synchronized. This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

Libraries and standards

There exists a wide range of primitives for message-passing. Many commercial software products (banking, payroll, etc., applications) use proprietary primitive libraries supplied with the software marketed by the vendors (e.g., the IBM CICS software which has a very widely installed customer base worldwide uses its own primitives).

The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community, but other alternative libraries exist.

Commercial software is often written using the remote procedure calls (RPC) mechanism in which procedures that potentially reside across the network are invoked transparently to the user, in the same manner that a local procedure is invoked.

1.6 Synchronous versus Asynchronous Executions

An asynchronous execution is an execution in which,

(i) there is no processor synchrony and there is no bound on the drift rate of processor clocks,

(ii) message delays (transmission + propagation times) are finite but unbounded,

(iii) there is no upper bound on the time taken by a process to execute a step.

An example asynchronous execution with four processes P0 to P3 is shown in Figure. The arrows denote the messages; the tail and head of an arrow mark the send and receive event for that message, denoted by a circle and vertical line, respectively. Non-communication events, also termed as internal events, are shown by shaded circles.

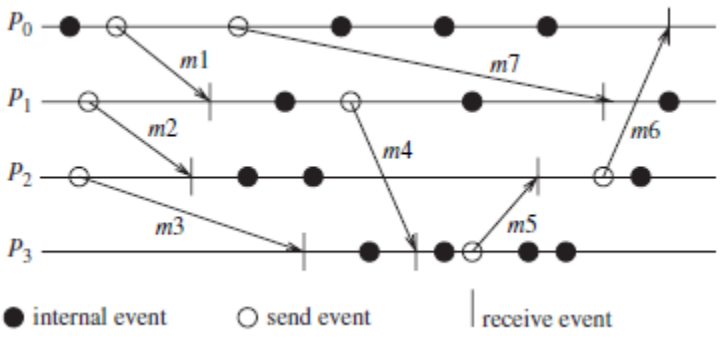

Figure: An example of an asynchronous execution in a message-passing system

A synchronous execution is an execution in which

(i) processors are synchronized and the clock drift rate between any two processors is bounded,

(ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round,

(iii) there is a known upper bound on the time taken by a process to execute a step.

An example of a synchronous execution with four processes P0 to P3 is shown in Figure. The arrows denote the messages.

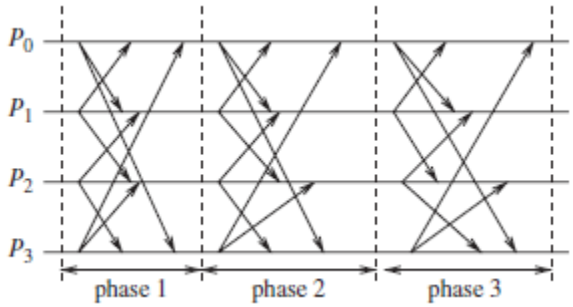

Figure: An example of a synchronous execution in a message-passing system

The synchronous execution is an abstraction that needs to be provided to the programs. When implementing this abstraction, observe that the fewer the steps or "synchronizations" of the processors, the lower the delays and costs. If processors are allowed to have an asynchronous execution for a period of time and then they synchronize, then the granularity of the synchrony is coarse. This is really a virtually synchronous execution, and the abstraction is sometimes termed as virtual synchrony.

Ideally, many programs want the processes to execute a series of instructions in rounds (also termed as steps or phases) asynchronously, with the requirement that after each round/step/phase, all the processes should be synchronized and all messages sent should be delivered. This is the commonly understood notion of a synchronous execution. Within each round/phase/step, there may be a finite and bounded number of sequential sub-rounds (or

subphases or sub-steps) that processes execute. Each sub-round is assumed to send at most one message per process; hence the message(s) sent will reach in a single message hop.

Emulating an asynchronous system by a synchronous system (A→S)

An asynchronous program (written for an asynchronous system) can be emulated on a synchronous system fairly trivially as the synchronous system is a special case of an asynchronous system – all communication finishes within the same round in which it is initiated.

Emulating a synchronous system by an asynchronous system (S →A)

A synchronous program (written for a synchronous system) can be emulated on an asynchronous system using a tool called synchronizer.

Emulations

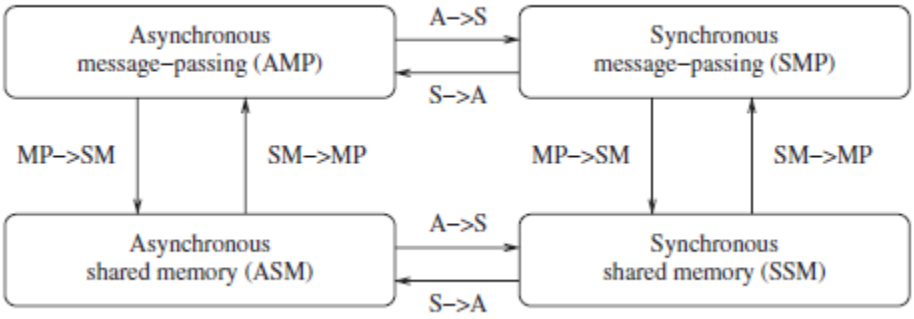

Figure: Emulations among the principal system classes in a failure free system.

There are four broad classes of programs, as shown in Figure. Using the emulations shown, any class can be emulated by any other. If system A can be emulated by system B denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A. Likewise, if a problem is solvable in A, it is also solvable in B. Hence, in a sense, all four classes are equivalent in terms of "computability" – what can and cannot be computed – in failure-free systems.