Realizing Programming Models in a Scalable Machine

[§7.2] Now let’s consider how programming models can be implemented on large distributed-memory machines.

- In a bus-based machine, LOAD and STORE are implemented by bus transactions.
- In parallel machines, network transactions are used instead.

**Network transactions**

A *network transaction* is a one-way transfer of information from an output buffer at the source to an input buffer at the destination. It causes an action at the destination that is not directly visible at the source.

The action may be simple ... writing the data to some location or complex ... executing a message-handler routine.

The effects of a network transaction are observable only through additional transactions.
Bus transactions

Let’s first review what goes on when an operation is performed over a bus.

The format of information in a bus transaction is determined by the physical wires of the bus—

- the data lines,
- the address lines,
- the command lines.

The information to be placed on the bus is kept in output registers until the bus is obtained.

A bus transaction begins with arbitration.

Bus-arbitration algorithms are

- usually implemented in hardware
- allow arbitration for one bus cycle to be overlapped with previous transfer.

Let us assume a static-priority algorithm.

- In case of a conflict, the higher-priority device gets the bus.
- Priorities do not change over time.
- Arbitration is implemented by means of daisy chaining.

  - The device closest to the central bus controller has the highest priority.
  - Requests are made on a common bus-request line.
  - The central bus-control unit propagates a bus-grant signal (BGT) if the bus is idle (as indicated by the acknowledge signal (SACK)).
  - When a non-requesting device receives a bus-grant signal, it merely sends it along to the next device.
When a requesting device receives a bus-grant signal, it asserts bus-busy, and does not send bus-grant along to the next device.

All modules examine the request, and one responds to the transaction.

Each request, then, is followed by a response.

- For a read, the response is the data plus a completion signal.
- For a write, the response

Buses in multiprocessors are often split-transaction buses, which divide a transaction into two subtransactions, request and response. Why?

Why are split-transaction buses susceptible to deadlock?

Either enough resources must be provided for the worst case, or a NACK signal must be provided.

**How network transactions differ**

They differ in protection; in a bus-based system, a processor performs all protection checks before putting a transaction on the bus.

In a scalable system, a component will often perform checks on incoming network transactions to help insure network integrity.
There is no single point of arbitration; many transactions can be going on simultaneously. This can cause deadlock.

Also, buffering is more complex, since transactions may be arriving from many sources at the same time.

**Shared address-space realization**

In a shared address-space system, a global address is usually composed of a module number and a local address:

```
module #    local address
```

However, instead of a module number, an address may specify a *route* to the module.

A shared address-space communication abstraction has these steps:

```
(1) Initiate memory access
(2) Address translation
(3) Local/remote check
(4) Request transaction
(5) Remote memory access
(6) Reply transaction
(7) Complete memory access
```

Requests may come in too fast for the destination node to process them. Why is this a problem in a network but not in a bus-based multiprocessor?

How is this problem manifested?
We can deal with this in two ways.

- Block requests when there is no buffer space. In this case,
- Drop requests. This requires some mechanism for retrying

In either case, a node must be capable of accepting requests and replies even when it cannot inject its own request. Otherwise, what can happen?

Coherence can also be a problem, and we will cover that later. But for now, let’s assume there are no caches.

**Achieving sequential consistency**

Achieving sequential consistency is harder than in a bus-based design. Why?

Write latency is large, so it is tempting to try to issue multiple write transactions without waiting for completion acknowledgments.

But this can cause problems.

In our example, suppose that $A$ and $flag$ are allocated in two different nodes.

![Diagram showing the program execution and data communication]

What can happen to lead to incorrect results (as shown in the diagram)?
How can this happen?

A correct implementation will wait for the write to \texttt{A} to be completed before issuing the write to \texttt{flag}.

**Message-passing realization**

Message passing may be done synchronously or asynchronously.

**Synchronous**

If it is done synchronously, the sending processor cannot know that the receive has completed until it receives another transaction. Thus, a three-phase protocol is required.
1. The *send* operation causes a “ready-to-send” message to be transmitted to the destination, carrying the source process and tag information.

When the remote process receives this, it checks a local table to determine if a matching *receive* has been performed by this process.

- If not, the ready-to-send information is recorded in a table to await the matching receive.
- If so, a ready-to-receive response transaction is generated.

The *receive* operation checks this table too.

- If a matching *send* has not been performed, the ready-to-receive information is reported.
- If a matching *send* is present, the receive generates a ready-to-receive transaction.

2. The sender waits until a “ready-to-receive” has arrived.

3. The data is transferred. The *receive* will complete when the data has been transferred.

Note that ready-to-send and ready-to-receive are small, fixed-format packets, whereas the data is a variable-length transfer.

*Asynchronous*

Assume now that message passing is done asynchronously. A simple implementation is shown in the diagram below.

- The *send* operation transmits a single block containing length information, and the identity of the sending and receiving processes.
- The destination strips off the “envelope” information and checks to see if a matching *receive* has been posted.
  - If so, it delivers the data to the specified address.
  - If not, it stores it in a temporary buffer.
There are some problems with this protocol.

- The data is streaming in while the receiving processor is trying to figure out where to put it. In order to figure this out, it needs to examine the process and tag information and consult the match table.

  Meanwhile, the destination processor has no idea where to put the data.

  *We could* allocate a temporary input buffer and then copy the information. What’s wrong with this?

- Second, we could run out of buffer space at the destination (just as in the SAS model).

  However, the problem is greater because messages are typically much larger.

  This could lead to deadlock.

The solution is to use a three-phase protocol similar to the synchronous case.

In this protocol, the data is kept at the sending node until the destination can accept it.

The destination issues a ready-to-receive when either—

- it has sufficient buffer space, or
• a matching receive has been executed

(1) Initiate send
(2) Address translation on $R_{dest}$
(3) Local/remote check
(4) Send-ready request
(5) Remote check for posted receive (assume fail); record send-ready
(6) Receive-ready request
(7) Bulk data reply
  Source VA $\rightarrow$ Dest VA or ID

To optimize for short messages, where the handshake protocol would dominate the time, a “credit” scheme can be used.

• Each process sets aside a certain amount of space for each process that might send it short messages.
• When a short message is sent, the sender deducts the length from its credit for that destination, until it receives an ACK that the message has been received.

In this way, short messages can usually be sent without a full handshake.

In summary, a three-phase protocol is usually required, but an optimistic single-phase protocol can be employed safely with a scheme for flow control.