26.7. Interprocess communication run-time support

In this Section we study a run-time implementation model for distributed memory machines, which

- exploits the features of the interconnection technologies described in Sections 26.4, 26.5, 26.6,
- exploits the run-time support optimizations studied in Section 23 according to the real
  architectural features and performance problems studied in Sections 17-21.

Let us consider two communicating processes allocated on two PEs belonging to Host machines H1
and H2, which don’t share memory.

The run-time support for distributed memory machines has two general characteristics:

1. the sender process in H1 delegates the (majority of the) send execution to Host H2, passing the
   interprocess message value, channel identifier and other parameters through the NIC and the
   Switch interconnect. In H2 the delegated send is executed according to the local run-time
   support, notably shared memory multiprocessor run-time support. The receive execution is local
   to H2, and a proper notification is sent to H1;

2. the final effect of an interprocess communication is a memory-to-memory copy of the message
   in H1 memory into the target variable in H2 memory (plus synchronization and low-level
   scheduling actions), though such copy doesn’t exploit a physically shared memory:

Delegation of send execution to H2.

Execution of the delegated send as a local send
(multiprocessor run-time support).

Local execution of receive.
26.7.1. Architectural features and NIC

In general, Hosts have a shared-memory multiprocessor architecture (single-CMP or multiple-CMP). A *Network Interface Card (NIC)* is connected to an I/O Interface (I/O INF) of a CMP:

![Diagram of Architectural Features and NIC](image)

NIC contains the logic (*Switch Interface*) for interfacing the Host to/from the *Switch Interconnect* of the distributed-memory architecture.

In traditional interconnect technology (e.g. 10 GbE), NIC is not a PE in the real sense of the word, instead it is a (set of) unit(s) specialized for Switch interfacing. In other words, it is similar to a classical I/O unit (e.g., disk unit) able to implement efficient DMA transfers using proper buffering and dedicated processing resources, but with limited programming capabilities.

From a logical point of view, the NIC can always be considered as a true PE (or subset of PEs) of the Host. As seen in the previous Sections, this is actually the real situation with *InfiniBand* and *Myrinet*-like technologies: in these cases,

- NIC is a programmable PE containing CPU(s), caching hierarchy, and possibly a local main memory;
- NIC is able to communicate with any ‘central’ PE, and to transfers data to/from the shared main memory M via DMA (see figure above). Such communications, with central PEs and with M, can be implemented in pipeline (Sections 10.5, 18.2.2, 21): in this way, a *wormhole* Switch Interconnect technology is exploited at best;
- the *Switch Interface* of NIC is a *local I/O unit* (see Section 17.4.2), optimized for the communications to/from the Switch Interconnect:
the interactions of NIC with central PEs are not merely limited to typical I/O interprocessor communications (e.g., parameters to start a communication with another Host) and DMA to/from main memory. Notably, also Cache-to-Cache (C2C) communications can be supported: in this way, pipelined data packets can be read/written from/to the central PE caches into/from the NIC caches and then to/from the Switch interconnect:

To reinforce this technological view, in some Network Processor CMPs (Section 16.1.5) the ‘smart’ NICs are contained in the chip itself, and cooperate with the central PEs and MINFs on-chip using the same internal network(s) and exploiting the general concurrency mechanisms.

More than one NIC can exist for the same CMP, or for multiple CMPs of the same Host, in order to optimize latency and bandwidth of interprocess communications. In general, for any inter-Host communication, a NIC scheduling algorithm in the run-time support provides to select a pair of NICs (the one in the sender, the other in the receiving Host) according to a load balancing strategy.
The space of possible solutions for run-time support design can be summarized as follows, where the solid lines correspond to current enterprise solutions, while the dotted lines are research proposals or are emulated for Ethernet interoperability:

We study the solution ‘smart’ NIC – primitive protocol – user mode in detail.

The adoption of traditional NIC, of IP protocol (both with ‘smart’ and traditional NIC), and of kernel mode will be evaluated as well by difference.

See also the considerations about performance portability in MIMD architectures (Section 15.5).

26.7.2. Rdy-Ack run-time support

Without losing generality, we refer to two LC communicating processes, P and Q, with a symmetric channel CH of asynchrony degree $k$ and message length $L$. Let $\text{PE}_P$ and $\text{PE}_Q$ the respective processing elements belonging to Host machines H1 and H2 respectively.

Let’s assume exclusive mapping (variations for multiprogrammed mapping will be mentioned) and ‘smart’ NICs (variations for traditional NICs will be mentioned).

Possible signatures of the communication libraries are:

\[
\begin{align*}
\text{send} & \quad (\text{MSG\_address}, \text{CH\_ID}, \text{L}, \text{Host\_dest}, \text{PE\_dest}) \\
\text{receive} & \quad (\text{CH\_ID}, \text{Host\_source}, \text{PE\_source}) \quad \text{returns} \quad (\text{VTG\_address}, \text{L})
\end{align*}
\]

The Rdy-Ack run-time data structures and algorithms are properly decentralized (partitioned) between $\text{PE}_P$ (H1) and $\text{PE}_Q$ (H2), and involve two NICs, the one in H1 (NIC1) and the other in H2 (NIC2).

**RTS(send): the role of sender node**

P virtual memory contains the private data structure $(\text{ACK}[k], \text{index})$: the array of Acks is used in a round-robin fashion, the current Ack being in the indexed position.

This is the channel descriptor partition in the sender node (CH_sender):
$RTS(send)$ tests $ACK[index]$: if false $P$ enters the busy waiting state with asynchronous wait (in multiprogrammed mapping, $P$ is suspended and a context-switch of $PE_P$ occurs), otherwise the $send$ execution is delegated to NIC1 and $index$ is updated.

The delegation is realized by an interprocessor message from $PE_P$ (through $UC_P$) to NIC1 (through $UC_{NIC1}$), fired by a sequence of Memory Mapped I/O Stores containing $(send\_type, MSG\_address, CH\_ID, index, L, H2\_id, PE_Q\_id)$ (and $P_id$ in multiprogrammed mapping).

NIC1 generates (through Memory Mapped I/O Stores) a streamed firmware message $send\_msg$ for the Switch Interface, to be sent onto the Switch Interconnect, containing:

- Header (at least 2 words): $H1\_id$, $PE_P\_id$, $H2\_id$, $PE_Q\_id$, message_type = $send$, $fw\_msg\_length$
- Value ($L + 4$ words): $CH\_ID$, $index$, $L$, $P_id$, and $MSG\_value$.

Remember that now $L$ may be large (the maximum value of $L$ is imposed by the concurrent language/library tools).

The streamed $send\_msg$ ($fw\_msg\_length = L + c$ words, $c \geq 6$) is pipelined word by word, and in this form it is transmitted onto the Switch Interconnect.

$MSG\_value$ is read in pipeline by the C2 unit of NIC1. If $PE_P$ is the home of $MSG$ in $H1$, $C2\_NIC1$ sends a C2C request to $C2_P$, which returns $MSG\_value$ to $C2\_NIC1$ in pipeline: it is likely that $MSG\_value$ is in $C2_P$, otherwise $C2_P$ starts the $MSG\_value$ pipeline transmission from $M$. Alternatively, in the cache-optimized version, the delegation by $PE_P$ could consists in a flush into $C2\_NIC1$.

The net effect is that $send\_msg$ is transmitted in pipeline from $C2_P$ (or $M$), through the Switch Interconnect, until it reaches $NIC2$ in the destination host $H2$. Of course, according to the value of $L$, $send\_msg$ might be decomposed into packets (this feature doesn’t change the basic behavior, thus in the following we’ll assume one packet).

With a traditional NIC1, $MSG\_value$ is read from $M$ in DMA mode. In some implementations $MSG\_value$ is pipelined, while in other implementations it is buffered in NIC1 before to be transmitted onto the Switch interconnect. In the latter case the cache optimizations are not exploited, with a corresponding penalty in latency. A much serious penalty is caused by the additional copy of $MSG\_value$: in no way the interprocess communication can be zero-copy.
**RTS(send): the role of destination node**

The *channel descriptor partition in the destination node* (CH_dest) consists of \( k \) VTG_S structures, scanned in a round-robin fashion, a CH_P array of \( k \) pointers to VTG_S structures, and an *index* to CH_P positions. Each VTG_S contains *(RDY, VTG_value)*, i.e. it doesn’t contain the ACK field which belong to CH_sender.

CH_dest

The second part of *RTS(send)* is delegated to the destination node: we use the term *local send* in the destination node to denote this part of *RTS(send)*.

In the most efficient implementation, *NIC2 itself provides to execute the local send*, according to the well-know run-time support scheme for a multiprocessor architecture.

The VTG_S array is shared between Q (the destination process) and the NIC2 process (i.e the process executed by NIC2, shortly NIC2 in the following).

Notice that NIC2 shares *all* those channel descriptors in H2 having a remote sender, thus such data structures must be allocated dynamically in NIC2 virtual memory (capability addressing technique, Appendix section 7).

NIC_2 receives the streamed *send msg* from the Switch interconnect, through the local Switch Interface, *in pipeline*, and in this form the *send msg* words are used to execute the local send: that is, the pipeline (which started in C2r) continues in H2 uninterruptedly until the local send ends, so a true zero-copy communication is implemented even in a distributed memory architecture:

Once the parameters CH_ID, index and \( L \) are received, *CH_dest address is determined as a function of CH_ID*. NIC2 performs the message copy of the received stream of \( L \) MSG_value words into the indexed VTG_S. The RDY field is updated.
If PE\(_Q\) is \textit{home} of CH\(_{\text{dest}}\), NIC2 executes the message copy (and RDY update) by a C2C write-back (local copy and Store notification) or, better, by a \textit{flush} into PE\(_Q\) caches.

If Q was in waiting state, it is waked up with an \textit{interprocessor notification} from NIC2 to PE\(_Q\) corresponding to the asynchronous wait (in a multiprogrammed mapping, the interrupt Handler in PE\(_Q\) puts Q_PCB into the respective Ready List).

A traditional NIC2 provides just to input the \textit{send_msg} words, while the \textit{local send} is delegated to PE\(_Q\) (or to another PE of H2):

- MSG\(_{\text{value}}\) is buffered by NIC2 into main memory or NIC memory,
- NIC2 interrupts PE\(_Q\), transmitting CH\(_{\text{ID}}\), L, and the identifier (capability) of the buffered message value;
- the interrupt \textit{Handler} in PE\(_Q\) executes the \textit{local send} by copying the message from M or from NIC2 memory (Memory Mapped I/O).

Therefore, at least another message copy is done and the interprocess communication cannot be zero-copy. Moreover, since the latency is increased (a PE (PE\(_Q\)) is interrupted to execute the local send), a communication processor is even more useful.

\textbf{RTS(receive)}

\textit{RTS(receive)} is executed by PE\(_Q\) locally operating on CH\(_{\text{dest}}\).

For \textit{set_ack}, a (set\_ack\_type, CH\(_{\text{ID}}\), index) notification is sent to PE\(_P\) via NIC2 (interprocessor communication from PE\(_Q\), Switch Interconnect, NIC1 (interprocessor communication to PE\(_P\)). The firmware message is:

- Header (at least 2 words): H2\(_{\text{id}}\), PE\(_Q\)\(_{\text{id}}\), H1\(_{\text{id}}\), PE\(_P\)\(_{\text{id}}\), message\(_{\text{type}}\) = \textit{set_ack}, fw\_msg\_length
- Value (3 words): CH\(_{\text{ID}}\), index, P\(_{\text{id}}\).

PE\(_P\) updates ACK[index] in CH\(_{\text{sender}}\)[CH\(_{\text{ID}}\)] and, if P was waiting, it is waked up.

\textbf{Send cost model}

The set up phase consists of the local manipulations of RDY-ACK and in the interprocessor communications inside H1 and H2. Using the \textit{primitive firmware protocol for the Switch Interconnect} and operating in \textit{user mode}, \(T_{\text{setup}}\) is of the same order of magnitude of \(T_{\text{setup}}\) for multiprocessors.

\(T_{\text{transm}}\) is the latency of the memory-to-memory (cache-to-cache) copy from H1 to H2. Using the primitive firmware protocol for the Switch Interconnect and operating in user mode, we have:

- firmware message \textit{send_msg} length: \(s = L + 6\);
- distance: \(d = 2 \times d_{\text{mp-interconnects}} + d_{\text{switch-interconnect}} + v\);

where \(v\) is the number of involved units inside the central PEs and NICs (\(v = 10\) in the previous figure example), and \(d_{\text{mp-interconnects}}\) is the total distance of the networks in the multiprocessor Host (single-CMP or multiple-CMP), including interface units (MINFs, WWs, I/O INFs);
\[ L_{\text{copy}}(L) = (s + d - 2)T_{\text{hop}} = (L + 2d_{\text{mp-interconnects}} + d_{\text{Switch-interconnect}} + w)T_{\text{hop}} \]

where \( w = v + 4 \), and \( T_{\text{hop}} \) is the Switch Interconnect one;

- \( T_{\text{transm}} = \frac{L_{\text{copy}}(L)}{L} \),
- \( L_{\text{com}}(L) \sim L T_{\text{transm}} = L_{\text{copy}}(L) \).

For single-rack clusters, the order of magnitude is the same of the multiprocessor architectures with comparable \( T_{\text{hop}} \).

It is worth noticing that the \textit{send} implementation scheme is not strictly dependent on the exploitation of C2C transfers. **If NIC is not able to act as a true cache-coherent PE**, the only difference is that NIC1 provides to start the pipelined stream of MSG\_value from M\(_1\) in DMA (instead of a C2C read from C2\(_P\), or a flush from C2\(_P\)), and NIC2 performs the pipelined DMA copy into VTG\_value in M\(_2\). The communication latency is roughly increased by

\[ 2 \left[ \frac{L}{\sigma_1} \right] \tau_M \]

if M\(_1\), M\(_2\) are the pipeline bottlenecks. However, the communication scheme is equally zero-copy.

The \textit{receive} effect is on \( T_{\text{setup}} \) only, while the \textit{set\_ack} communication occurs asynchronously and in parallel with the host processing activities, and has negligible impact on the base latency analysis.

If the \textit{IP protocol} is used for the Switch interconnect, the interprocess communication latency is increased by about one-two orders of magnitude, both in \( T_{\text{setup}} \) and in \( T_{\text{transm}} \), if the \textit{user mode} design is adopted, while it is increased by at least three orders of magnitude in \textit{kernel mode}.

### 26.7.3. Variants to run-time support design and under-load analysis

The run-time implementation of the previous Section is based on the \textit{send delegation to the destination Host}, i.e. delegation of \textit{send} execution to the Host in which the destination process is allocated.

A dual approach can be defined, namely **receive delegation to the sender Host**, which is characterized as follows:

- **CH\_sender**: each VTG\_S consists of ACK and
  - \textit{pointer (capability) to the message}, for zero-copy communication: valid if the message is not modified before the \textit{receive} execution,
  or
  - \textit{MSG\_value}, if an \textit{additional copy} of the message into this field is performed: valid if MSG can be modified before the \textit{receive} execution, and the message copy latency is substantially lower than the calculation time.

This structure, and the message in the former case, is \textit{shared} with NIC1;
• \textit{RTS}(send) is local: if and when ACK is true, the message or the message pointer is written in the indexed VTG\_S. A \textit{set\_RDY} notification (in particular containing PE\_dest, CH\_ID and the current index value) is sent, through NIC1 and the network, to H2. NIC2 updates the RDY field of the indexed VTG\_S in CH\_dest, and notifies to PE\_dest the process wake up if necessary;

• \textit{RTS}(receive): if and when RDY is true, a request is done to H1 (specifying, PE\_sender, CH\_ID, index). H1 (NIC1) replies with the message value to H2, where NIC2 copies the message value into the target variable: all these actions are performed in pipeline. When the communication to H2 is concluded, the ACK is updated locally in CH\_sender.

The receive-delegation approach is modeled as a client-server system with request-reply pattern, where actually clients and servers are NICs:

\begin{center}
\begin{tikzpicture}
\node[draw,rectangle] (H1) at (0,0) {H1};
\node[draw,rectangle] (H2) at (4,0) {H2};
\draw[->,blue] (H1) -- (H2);\node at (2,-0.5) {send is executed locally and a RDY is notified to the destination node};
\draw[<-] (H2) -- (H1);\end{tikzpicture}
\end{center}

\begin{center}
\textit{receive} consists of a request phase (give me the message) and of a reply phase (message value)
\end{center}

Therefore, the evaluation methodology of Sections 21.6, 21.7, 21.8 is valid. In particular, low-\textit{p} mappings could be achieved with a proper strategy of channel descriptors allocation.

For example, in a farm or data-parallel program (IN, \{Worker\}, OUT), each Worker node contains the channel descriptors from IN and to OUT.

Moreover, we recognize a substantial communication latency ($L_{com}$) increase compared with the send-delegation approach.

The send-delegation approach is equally modeled as a client-server system but \textit{without} request-reply pattern:

\begin{center}
\begin{tikzpicture}
\node[draw,rectangle] (H1) at (0,0) {H1};
\node[draw,rectangle] (H2) at (4,0) {H2};
\draw[->,blue] (H1) -- (H2);\node at (2,-0.5) {send is delegated to the destination node, where a local send is executed};
\draw[<-] (H2) -- (H1);\end{tikzpicture}
\end{center}

\begin{center}
\textit{receive} is executed locally and an ACK is notified to the sender node
\end{center}

In this case, there is no specific problem of low-\textit{p} mapping and channel allocation. In the (IN, \{Worker\}, OUT) example there is no convenience to allocating the Worker-OUT channel in the Worker node: \textit{receive} would be executed according to a request-reply pattern (notice the quite different situation with respect to a shared memory architecture).
The *channel allocation* strategy in the destination node does not introduce a request-reply contention problem.

However the multiple-client communications towards a single server (e.g. \{Worker\} towards OUT) introduce a *congestion problem* as well, though of different nature, i.e. the classical congestion problem of acyclic graph computations:

- the server (NIC of destination Host) could become a *bottleneck* (it could be $\rho > 1$ in the transient period), with a consequent, substantial increase of the client effective service time.

As usually, in such cases the solution consists in *increasing the server bandwidth*, notably by providing

- more than one NIC per Host,

and/or

- NICs with an internal parallel architecture (e.g., a dedicated multiprocessor).

Actually, we are implementing a *farm of NICs*, for which a *load-balancing* strategy (on-demand) must be provided in order to optimize the aggregate bandwidth:

The number of NICs is a critical issue. As known, it is given by the ratio between their ideal service time and the interarrival time: the former depends mainly on the interconnect technology and on the approach to the run-time support design, the latter depends on the clients bandwidth and on the network bandwidth. Thus, the configuration problem is not a simple one: a flexible approach to system configuration is needed for high-performance applications and their performance portability.