Reti attive

- **Parte II**, Docente: Ing. Andrea Santoro
  - Reti Attive
  - Studi di Caso (Myrinet)
  - Tecniche di Bypass del S.O.
  - Datagram e Streaming su Reti Programmabili
  - Studi di caso (GM, FastMessages 1.x, FastMessages 2.x)

Reti Passive (Passive Networks)

- Traditionally, the function of a communication network has been to deliver the packets from one endpoint to another.

Input

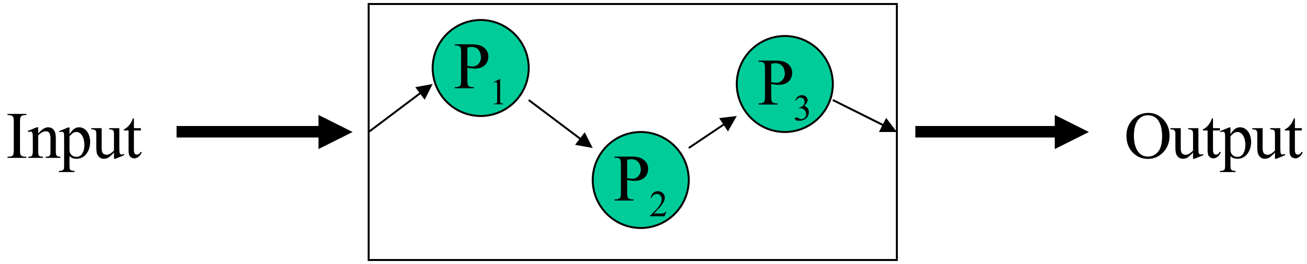
Output

Example: *ethernet*.

- Conventional routers just examine packet headers, e.g. the IP header, and forward packets according to the contents of a routing table. These “passive” routers switch the packets from one port to another without modifying the packet contents. Passive router functions are limited to the Network and Transport Protocol layers.

Reti Attive (Active Networks)

- **Definition:** Active networks allow individual user, or groups of users, to inject **customized** programs into the nodes of the network.



Why?

- "Active" architectures enable a massive increase in the complexity and customization of the computation (routing, flow control, error correction, etc…) that is performed within the network, e.g., that is interposed between the communicating end points.

More in depth…

- **Active networks** are characterized by dynamic programmability where the users can program the network hardware to manipulate the information flowing through them, which might lead to a change in transport system behaviour.

- The network is active in the sense that it can dynamically perform computation on the user data, while packets can carry information and/or program to be executed by intermediate nodes and possibly change their state.

Advantages (I)

- **Flexibility**: Active networks provide a much more flexible network infrastructure with increased capabilities compared to traditional passive networks.
- **Optimization**: redundant controls executed in different places of the protocol stack can be condensed a single time.

Advantages (II)

- Routers in **passive networks** can only perform computation up to the network layer: computation in the application layer is carried out by 'smart' hosts sitting at the edge of the network.
- It is difficult to integrate new technologies and standards into the traditional network infrastructure. Redundant operations at several protocol layers lead to poor performances and difficulties in accommodating new services that might require computation within the network such as firewalls, web proxies, mobile proxies, etc.

Active Networks Approaches

- Active packets
- Active nodes
- Hybrid

Active Packets - definition

- Most common approach to early active network architecture.

- The executable code is carried in the active packets and no code resides in the nodes. The nodes are active because they allow computation to take place up to the application layer.

Active Packets – Drawbacks

- Performance related problems because of the huge safety and security requirements. This stems more from the fact that users can define arbitrary functions to be computed.
- It also suffers from capability related problems because the only way to minimize the security and safety issues is by restricting the programs carried in the packets.
- Increased bandwidth and/or latency consumption to carry the programs inside the packets.

An Example

- An example of this architecture is the *smart packets* approach as proposed by BBN technologies. They decided that programs must be self-contained and must fit entirely into one packet, so programs cannot be more than 1kbyte. They also made the operating environment provide safety and security because packets containing executable codes are extremely dangerous. A way of achieving this is to check whether a program comes from an authorized user, and also to check its integrity in each node.

- Still…. What happens if the authorized user is malicious? Or just makes an honest mistake?

Active Node - Definition

- The active packet does not carry the actual code, but they carry some identifiers to predefined functions that reside in the nodes. The packets decide which function is going to be executed on their data, and also provide the parameter for these functions. But the actual code for these functions resides in the active node.

Active Networks - Problems

- In the active node approach, users cannot define arbitrary functions, so it has an advantage with respect to security and efficiency.
- However, it is slightly restricted because only the functions that have been preloaded can be called upon.

Hybrid (both packets and nodes are active)

- This is a combination of the active packet and active node approaches. Here the active packets carry actual code which is relatively simple, and more complex code resides in the active nodes.

Active Networks Drawbacks (I)

- *More complex (expensive) hardware*

  Making nodes capable to run programs and at the same time process communication at a rate comparable with existing off-the-shelf passive networks requires more complex hardware and designs than those available in passive networks.

Active Networks Drawbacks (II)

- *Safety and Security*

  Giving a user the ability to program the network increases the security and safety problems. In general, the attacks that an active node is susceptible to are greater than in current passive networks. Some of the security problems could be:

  - misuse of an active network node by the active code
  - misuse of active code by other active code
  - misuse of active code by an active network node
  - misuse of active code by the underlying infrastructure

Active Networks Drawbacks (III)

- Transparency

  Passive networks are End-to-end. You give a packet (or stream) as input, you receive a packet (or stream) in output, and you don’t worry at all about anything in the middle.

  In active networks the “middle” may be important since the application using it is affected by the active code running on the network. Ensuring both transaperency AND flexibility is a complex issue.

An Example of Active Network: Myrinet

- Myrinet is a good example of high-performance, high-availability, cluster interconnect active network.

Why Myrinet?

- It is employed in many of the world's premier cluster-computing systems. A total of 141 (28.2%) of the June-2005 TOP500 supercomputers use Myrinet technology.

Myrinet Characteristics

- Myrinet is a cost-effective, high-performance, packet-communication and switching technology widely used to interconnect **clusters** of workstations, PCs, servers, blade servers, or single-board computers.
- Myrinet Clusters provide an economical way of achieving:
  - Computation distribution across an array of cost-effective hosts.
  - low-latency communication between host processes.
  - Fault detection and isolation
  - Alternative communication paths.

Characteristics distinguishing Myrinet from other (passive) networks

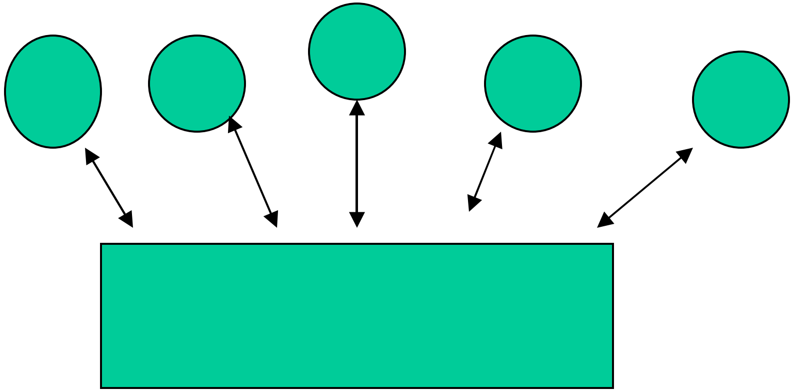
- Flow control, error control, and "heartbeat" continuity monitoring on every link.
- Low-latency, wormhole switches, with monitoring for high-availability applications.
- Switch networks that can scale to tens of thousands of hosts, and that can also provide alternative communication paths between hosts.
- **Network Interface Cards (NICs) that executes customized firmware to offload protocol processing from the host computer.**

How is Myrinet an “Active Network”?

- **Network Interface Cards (NICs) that executes customized firmware to offload protocol processing from the host computer.**
- The “Customized firmware” is very flexible and can effectively run any program of size up to 4-5 megabytes.

Myrinet Network Architecture

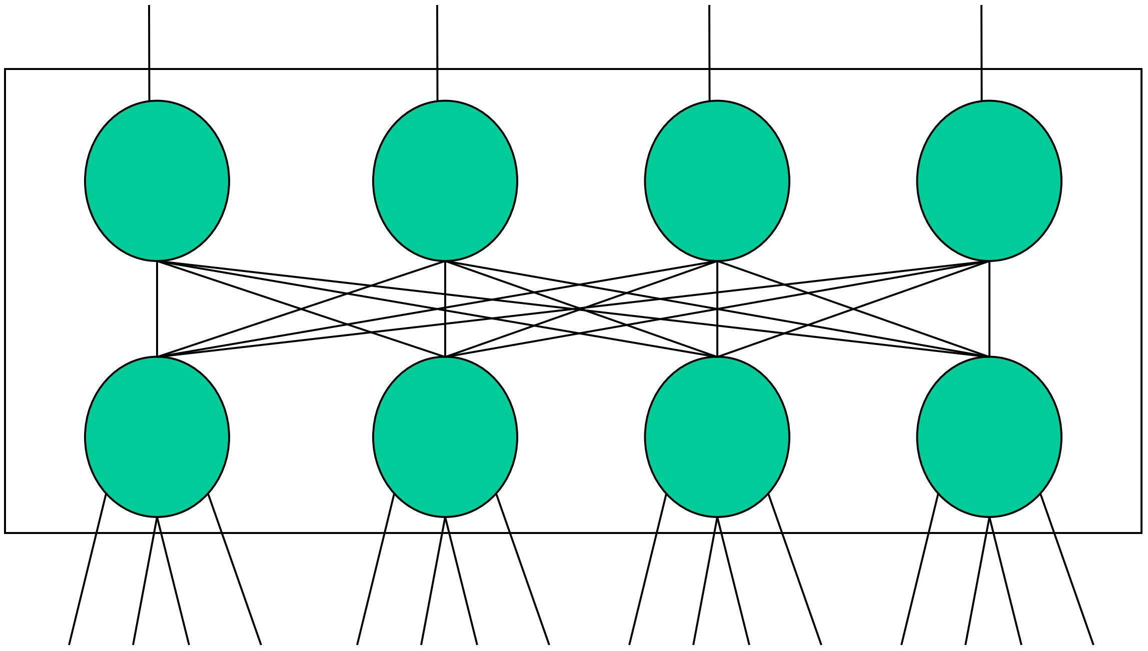
- Interconnection switch – wormhole technology
- Network Interface card (NIC), programmable from the user.
- **Full-duplex** interconnection wires – optical or copper technology.



Myrinet Switch Architecture

- A switch in a network is composed by several 8X8 **crossbars** (in some cases 16X16) organized in a fat tree.

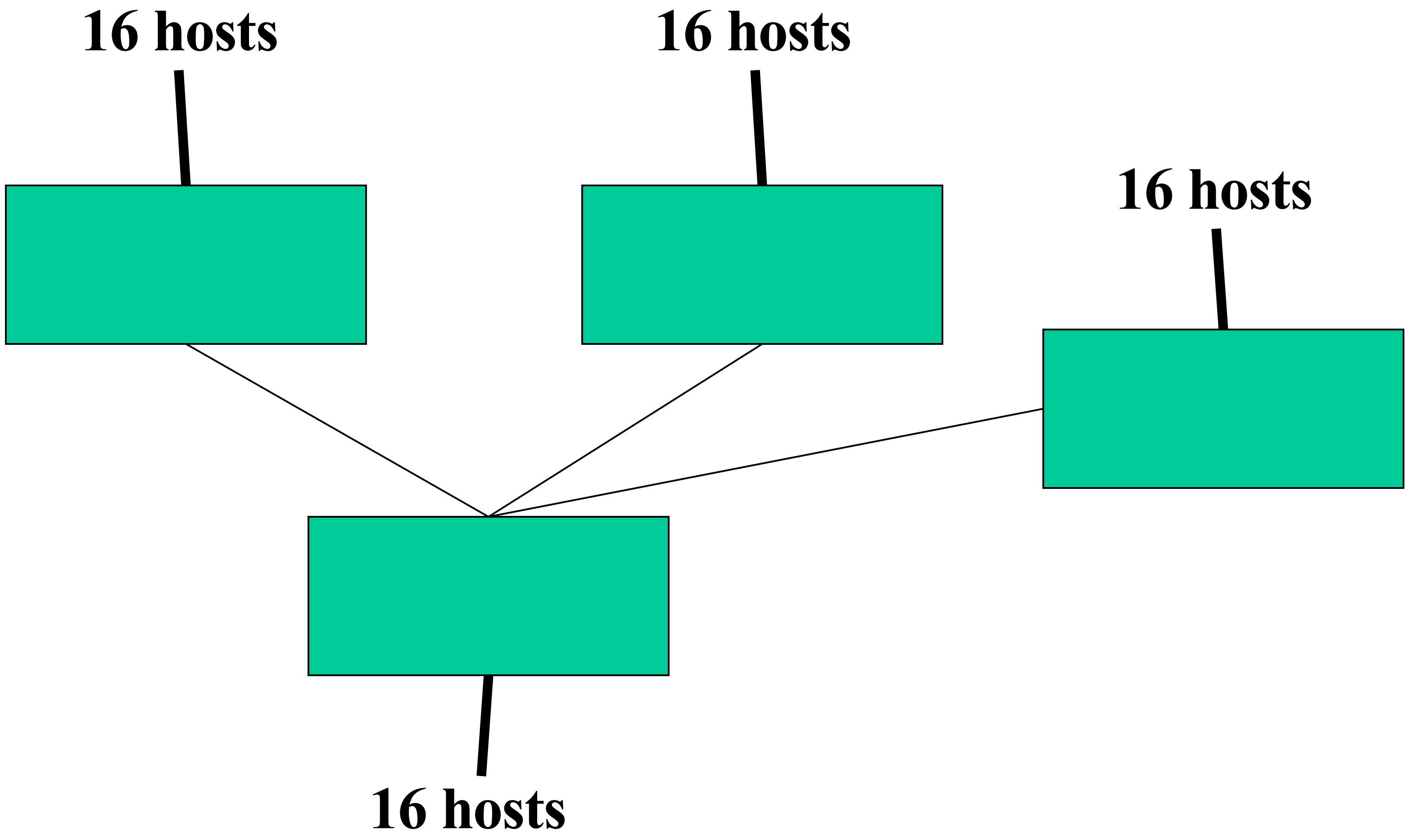
?????????



16 Hosts

Extending a Myrinet Network

"Switch networks that can scale to tens of thousands of hosts"

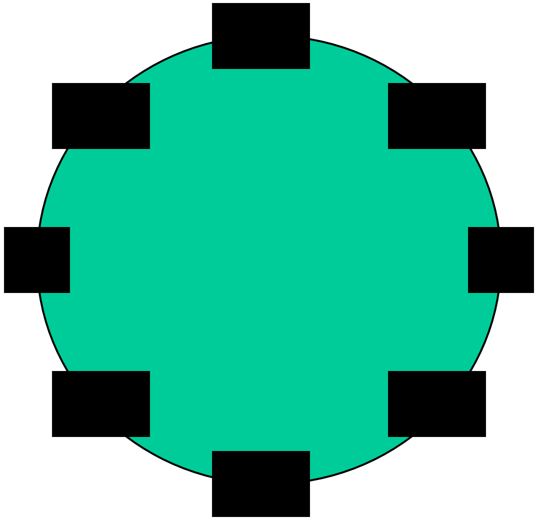


Examples of Myrinet Physical Layers

- Full Duplex
- SAN (System Area Network):
  - 10 signals full duplex.
  - 8 data bits
  - 1 control/data bit
  - 1 stop/go bit for flow control
- LAN (Local Area Network)
  - 9 signals full duplex (18 signals)
  - 8 data bits
  - 1 control/data bit
  - Control Flow realized with an additional control symbol

Routing in Myrinet

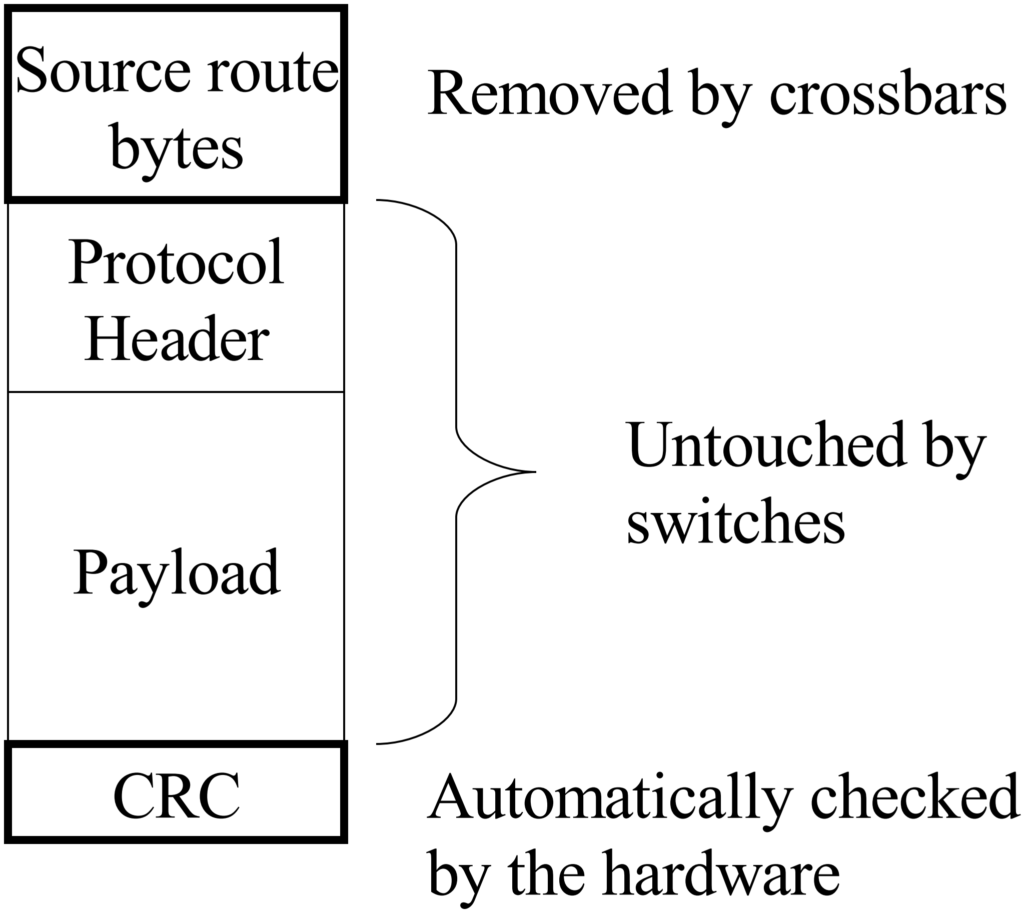
- Takes place at the crossbar level.



- First byte identifies the route that the packet must take, and is removed.

Source routing

Myrinet Data Link layer



Source route bytes

Removed by crossbars

Protocol Header

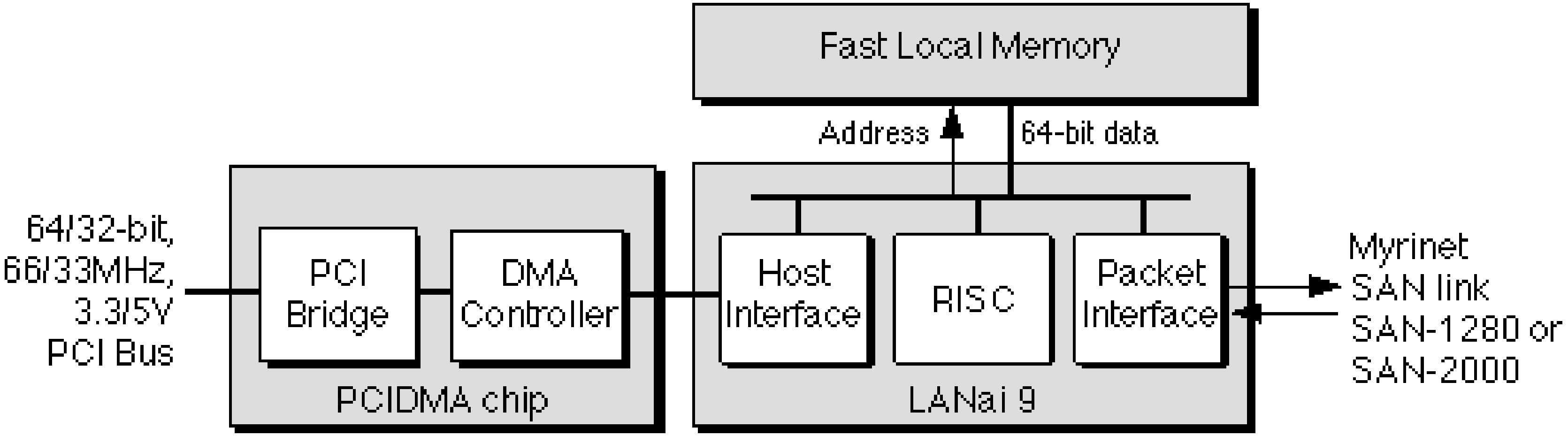
Payload

Untouched by switches

CRC

Automatically checked by the hardware

Myrinet NIC Architecture

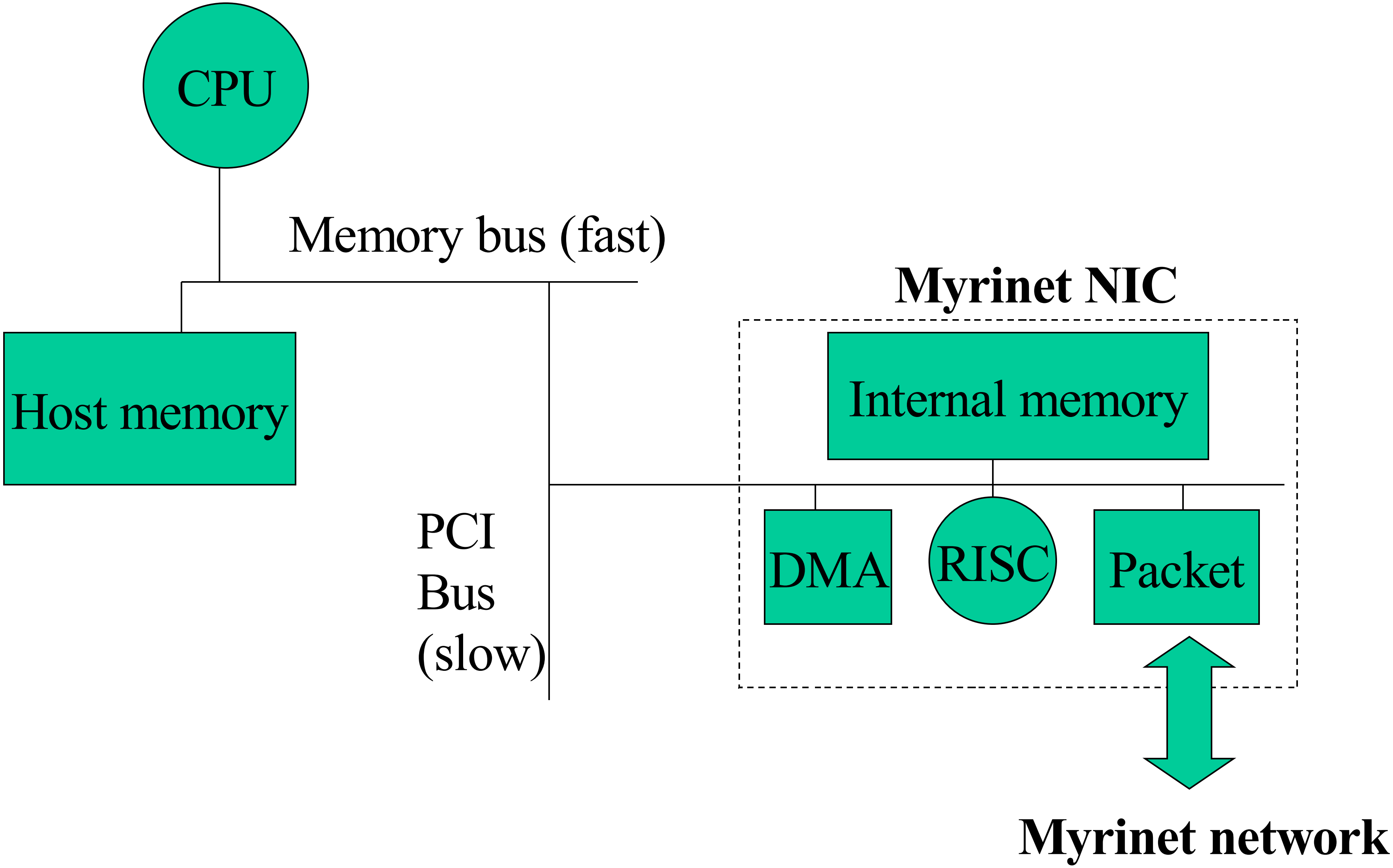


- Internal BUS
- Internal memory (mappable on the host)
- RISC Processor
- Packet Interface (RDMA and SDMA)
- External BUS (EBUS) DMA

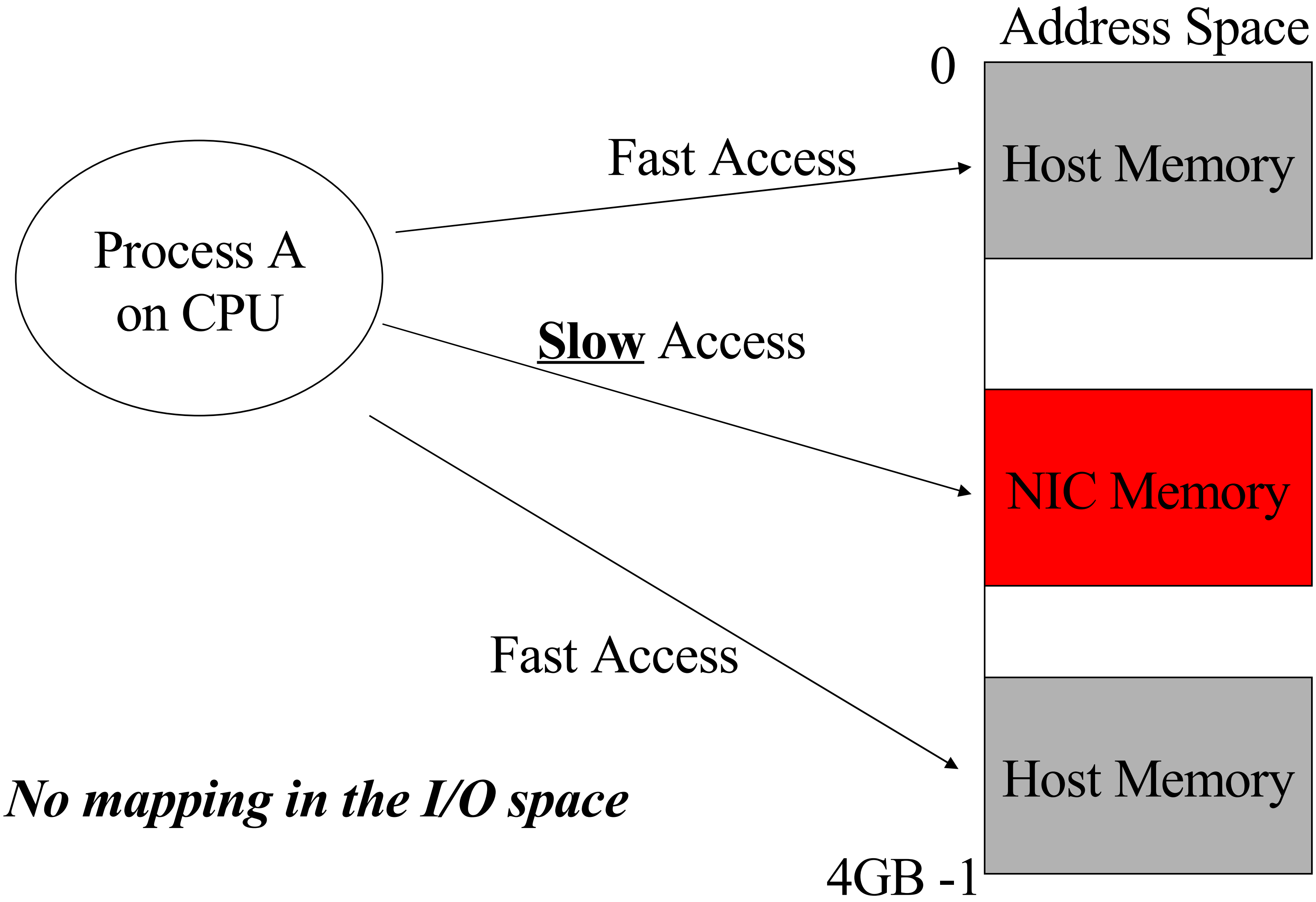
Myrinet Control Program (MCP)

- The Internal memory contains a program executed by the RISC processor. The RISC processor has full control on the packet interface and partial control on the EBUS DMA, thus it can manage packet send and receive, plus sophisticated analysis of incoming packets.
- Basically the MCP substitutes the firmware of a common network card.

Myrinet NIC in a typical PC

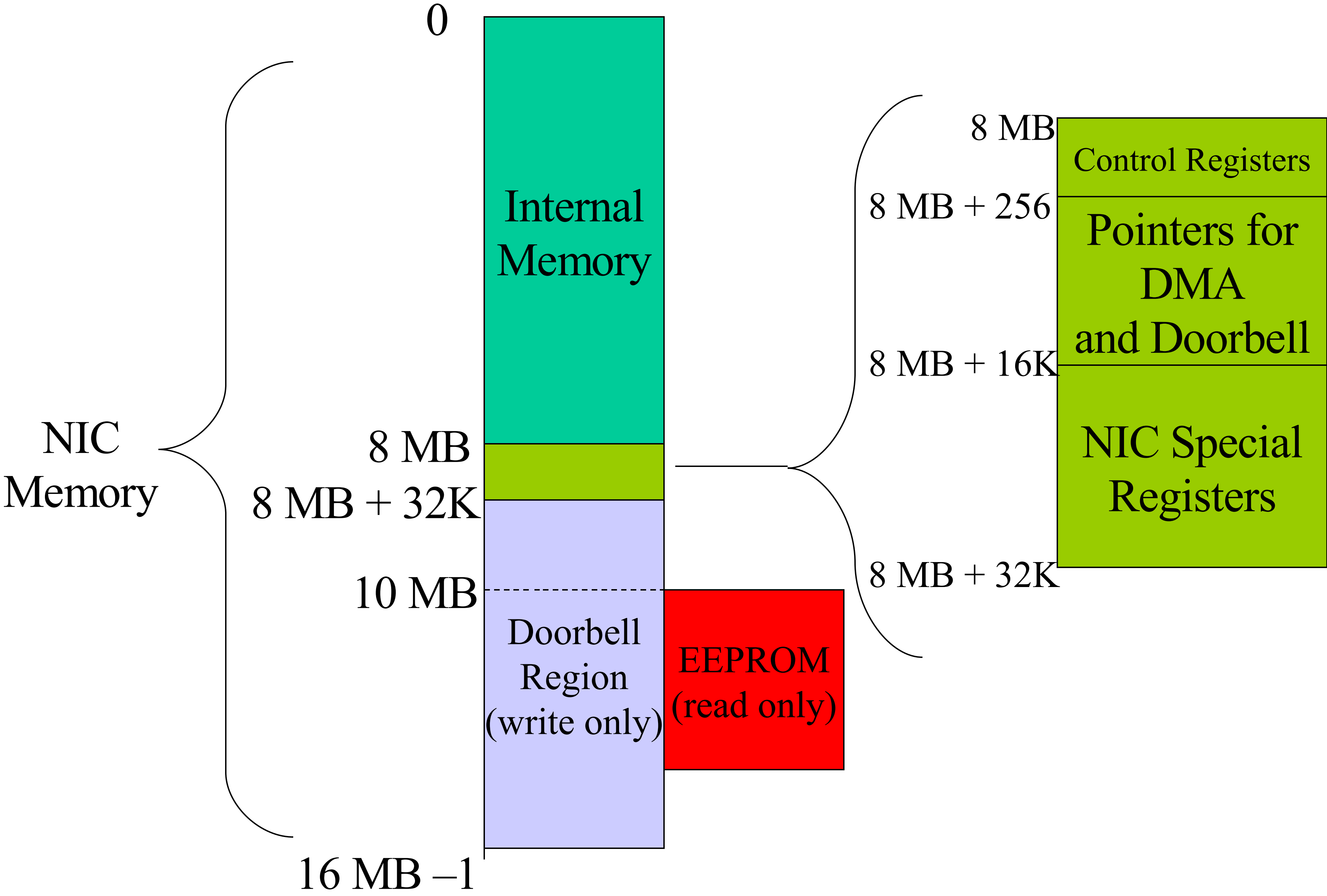


The OS point of view



No mapping in the I/O space

Expanding NIC Memory….



Myrinet NIC bootstrap

- Driver activates the external bus interface
- Driver internal maps memory into kernel memory space
- Driver loads (memcpy) MCP into NIC memory
- Driver tells RISC processor to start.

The Host processor

- Has control over:
  - External control registers
  - Internal Memory
  - DMA
  - Interrupt signals (partial)
- All interaction take the form of writing/reading into special memory addresses offset from the base pointer.

The LANai processor

- Has control over:
  - Control registers (partial)
  - Packet interface: RDMA/SDMA
  - Internal Memory
  - DMA (partial)
  - Interrupt signals (partial)
- All interaction take the form of writing/reading into special registers (mapped into dedicated memory addresses).

Technical specs

- PCI Bridge
  - EBUS DMA
  - Doorbell region
- RISC Processor
  - Special registers

DMA Controller (I)

- The EBUS DMA is controlled by four linked lists of operations.
- The elements in the list are structures that describe a single DMA operation.

```
struct DMA_BLOCK {
   volatile uint32 next; /* next block in chain pointer */
   uint16 spare; /* unused */
   uint16 csum; /* ones complement cksum of this block */
   uint32 len; /* byte count */
   uint32 lar; /* lanai address */
   uint32 eah; /* high PCI address - unused for 32-bit PCI */
   uint32 eal; /* low 32-bit PCI address */
};
```

DMA Controller (II)

- The four lists have different priorities (I.e. one list must be empty before an operation with lower priority must be executed)
- The DMA must be activated before it starts processing the operations in one of the lists.
- The four lists can be activated separately, or more than one at the same time.
- Once activated the DMA executes automatically EVERY operation in the activated lists, according to their priority. At the end of a list the terminated list is deactivated.

DMA Controller (III)

- A single DMA operation is NON-preemptable (an operation in progress is not stopped even if a higher priority list is activated)
- The pointers to the four lists are kept into four registers inside the DMA controller and can be accessed only by the host (i.e., by the OS driver).
- The structures describing the DMA operations are kept inside the internal NIC memory and can be written by both the host and the RISC processor.
- The host can activate all the DMA channels. The RISC processor can activate only two of the four channels.

Doorbell Region

- Doorbell is a mechanism to temporize FIFO writes inside the internal memory.
- Each time a write access is performed in the doorbell memory space an element of the following type is added to a list:

```
struct FIFO_ENTRY {
  uint32 addr;
  uint32 data;
};
```

Problems writing a MCP

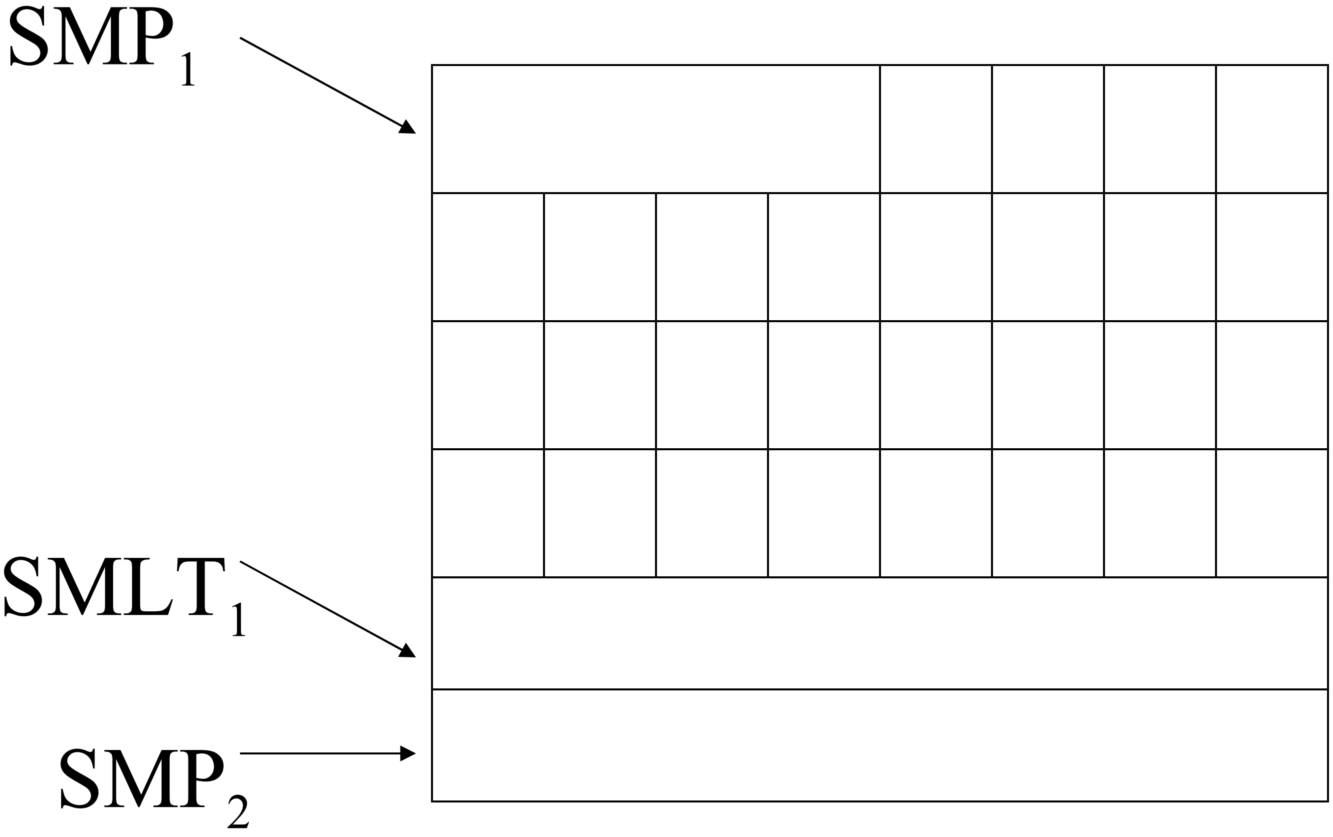
- MCPs are not written in assembly, but with a proper high-level language.
- Typical language is C, since it is a high-level language tha still allow full control of the memory layout.
- The MCP cannot be written by simply using the standard compiler, because it runs on a different architecture.
- This means (for instance):
  - Different instruction set
  - Different register organization (big-endian, little-endian)
  - Different memory layouts.

Cross-compiling

- Thus any MCP must be generated by a *cross-compiler*. It is a compiler that compiles code for a different architecture than that running the compiler.
- Particular attention should be done when passing data between the CPU and the NIC, since one of them might work in little-endian, while the other works in big-endian.
- Actual case: IA32 architecture is little endian, while myrinet is big endian.
- In this case “transparent conversion” functions such as “htonl()” ***MUST*** be employed. They work because *by definition* “network order” is always big-endian.

Send Registers

- SMP
- SA
- SMLT



SMP_1

$SMLT_1$

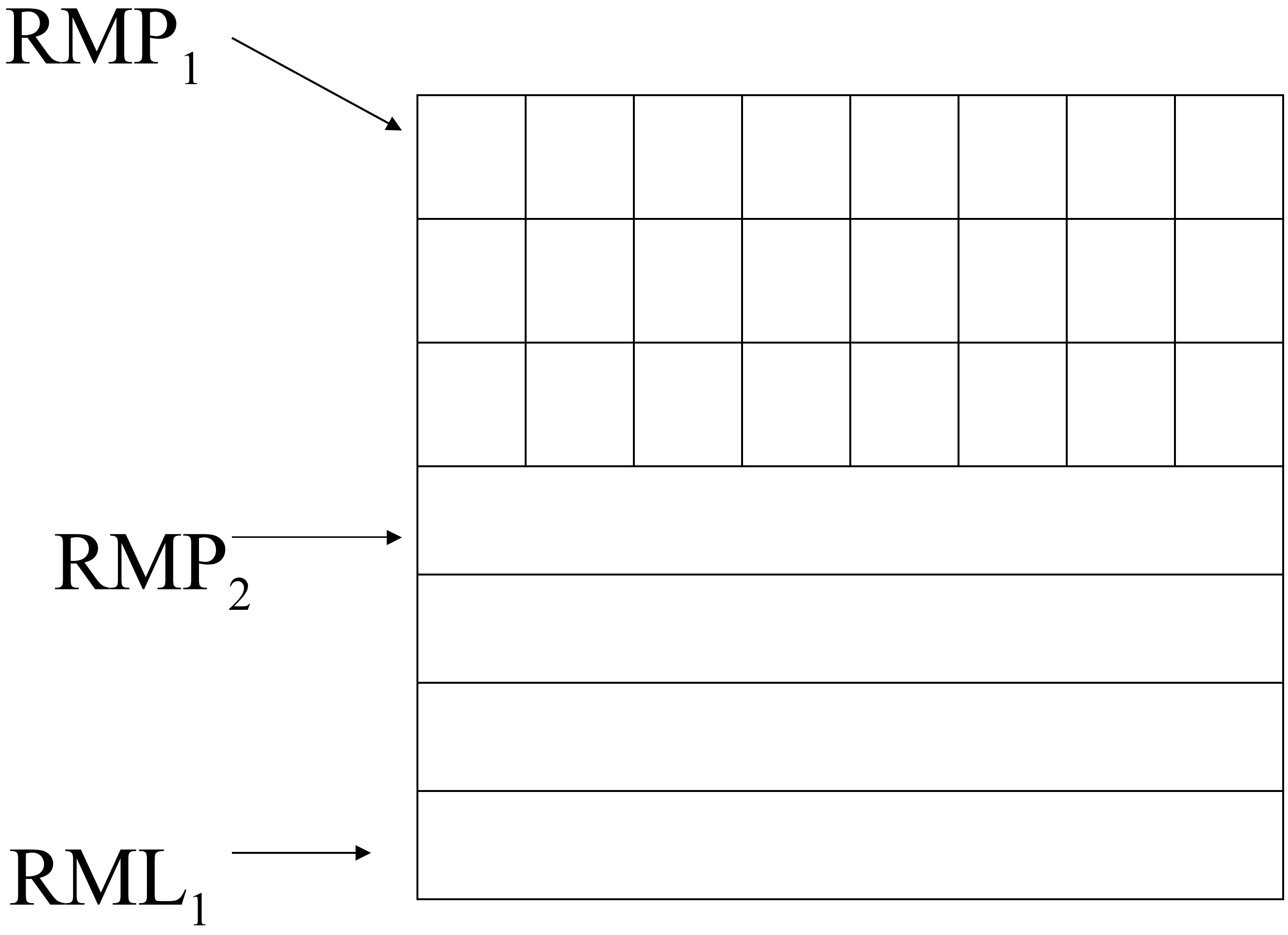
SMP_2

First 3 bytes of SMP and SMLT hardwired to zero

SA = 0x4

Receive Registers

- RMP
- RML



First 3 bytes of RMP and RML hardwired to zero

Status Registers

- ISR (Interface Status Register)

  Contains the chip status information. Readbale by NIC and host, partially writable by both.

- EIMR (External Interrupt Mask Register)

  When a bit of ISR AND the corresponding bit of EIMR are equal to 1 an interrupt is sent to the host CPU. Readable and writable by both NIC and host.

- PULSE

  Activates the DMA lines controllable by the processor. Writable by the NIC, unreadable.

Interface Status Registers (I)

- HOST_SIG

  Set to 1 by RISC processor, reset to 0 from the host. Used to send various signals/interrupts from the RISC to the host.

- LANAI_SIG

  Set to 1 by CPU, reset to 0 from the RISC processor. Used to send various signals from the host to the RISC.

- WAKE0_INT

  Set to 1 when at least 1 EBUS DMA transfer is completed. reset to 0 from the RISC processor by writing 1 into it. Used to send various signals from the host to the RISC.

Interface Status Registers (II)

- SEND_INT

  Set to 1 when the SDMA is completed. Set to 0 when the programmer writes 1 into it or a new SDMA is initiated.

- BUFF_INT

  Set to 1 when the buffer for the RDMA is exhausted. Set to 0 when the programmer writes 1 into it or a new RDMA is initiated.

- RECV_INT (External Interrupt Mask Register)

  Set to 1 when the RDMA of a packet has been completed. It is cleared when the programmer writes 1 into it.

Examples of using Status Registers

- ISR

  ISR = SEND_INT;

  Sets to 0 the SEND_INT bit in ISR

  If (ISR & SEND_INT) ….

  Tests the content of the SEND_INT bit

- EIMR

  EISR = 0x0;

  Disables any interrupt sent from the NIC to the CPU.

- PULSE

  PULSE = 0x2; PULSE = 0x4;

  Activate the DMA queue associated with the second or fourth DMA list, respectively.

Mapping

- The source routing requires every node to have knowledge of the whole network
- This is accomplished by having a portion of the MCP devoted to perform “mapping” of the whole network.
  - The simpler MCPs just have static mapping. The network is assumed to be reliable and mapping is performed only at initialization.
  - In case of unreliable networks the network mapping and new routes establishments must be performed on the fly.

Mapper

- The software employed to discover the map of network usually takes the name of “mapper”. It employs “scouting” packets to discover active links and nodes.
- Care must be taken when designing the routes among the nodes which might otherwise result in a deadlock.

Communication Techniques For Active Networks

Typical evaluation parameters in networking

- **Bandwidth**

  Amount of data sent (or received) over the network in given time. Typically relevant when evaluating the performance of peer-to-peer systems, WANs, etc…
- **Latency**

  Time required for a specific piece of information to go from the source to the destination. Typically relevant when evaluating the behavior of parallel simulators, or parallel computation applications with strict synchronization requirements.

Traditional Communication Techniques (I)

- User space is where all user programs execute. Historically, applications operating in user space make system calls into the kernel for privileged operations such as I/O commands to network or storage devices.

Examples for Communication:

- read()
- write()

Traditional Communication Techniques (II)

- Kernel space is where the operating system, device drivers and hardware interrupt handlers run. The kernel provides a safe interface to hardware, provides interprocess security, gives different processes a fair share of the resources, and arbitrates access to resources/hardware.

Traditional Communication Techniques (III)

- Transitions from user to kernel space (and the reverse) historically have been required to pass data between user programs and their clustering, storage and networking hardware resource. Each transition requires:
  - Spending time to transition between the user mode and the kernel mode.
  - Copying parameters from user space to kernel space.
  - Copying return information from kernel space to user space.

Traditional Communication Techniques (IV)

- When the traditional communications techniques were developed, communication times were much larger than the overhead of changing between the different execution modes and of saving and restoring context information.
- However nowadays this overhead limits application I/O performance. As mentioned before, in the case of TCP/IP user-to-kernel transition, it can account for approximately 40% of the host CPU networking overhead.

OS Bypass

- The technique for eliminating the user-to-kernel transition is known as Operating System bypass.
- Operating system calls are avoided by updating the I/O library to take direct control of the network card from the application.
- This usually requires that the network card driver exposes to the user level some portions of the network card hardware.
- This modification enables direct communication of all commands to the I/O adapter, eliminating the user-to-kernel transition.
- Operating system bypass is a well-proven technique used for years in the highest-performance cluster interconnects, such as myrinet.

Why it is not used for everyday communication?

- Protection and Security issues
- Special hardware required
- Transparency issues

Security (and protection) issues

- Interacting with the hardware leads to many security issues.
- A mistake can lead to severe consequences. Not simply a crash of the process, but hang the whole machine.
- Even in case of a correctly working library an attacker (or even a simple misplaced pointer) can touch some control register that shouldn’t be touched wreaking all sort of havoc.

Special hardware required

- It is not strictly needed.

  A correctly written driver might expose the hardware controls of the network card (such as DMA access) to the user level

- BUT: IS THAT A SECURE APPROACH???

- Obviously, giving the user control of (for example) the DMA without proper hardware limitations is a huge security risk.

Transparency issues

- The typical System Call mechanism allows the developer to treat the machine as a black box.
- Example: system call *write()*;
- Writing on a network card WITHOUT using system calls usually entails knowing hardware details.
- Partial solution: having a standard (user-level) library interfacing with the network card, that must be called instead of the system calls.
- Limitations:
  - Third party libraries who relies on standard system calls cannot be used.
  - Transparent, but not secure.

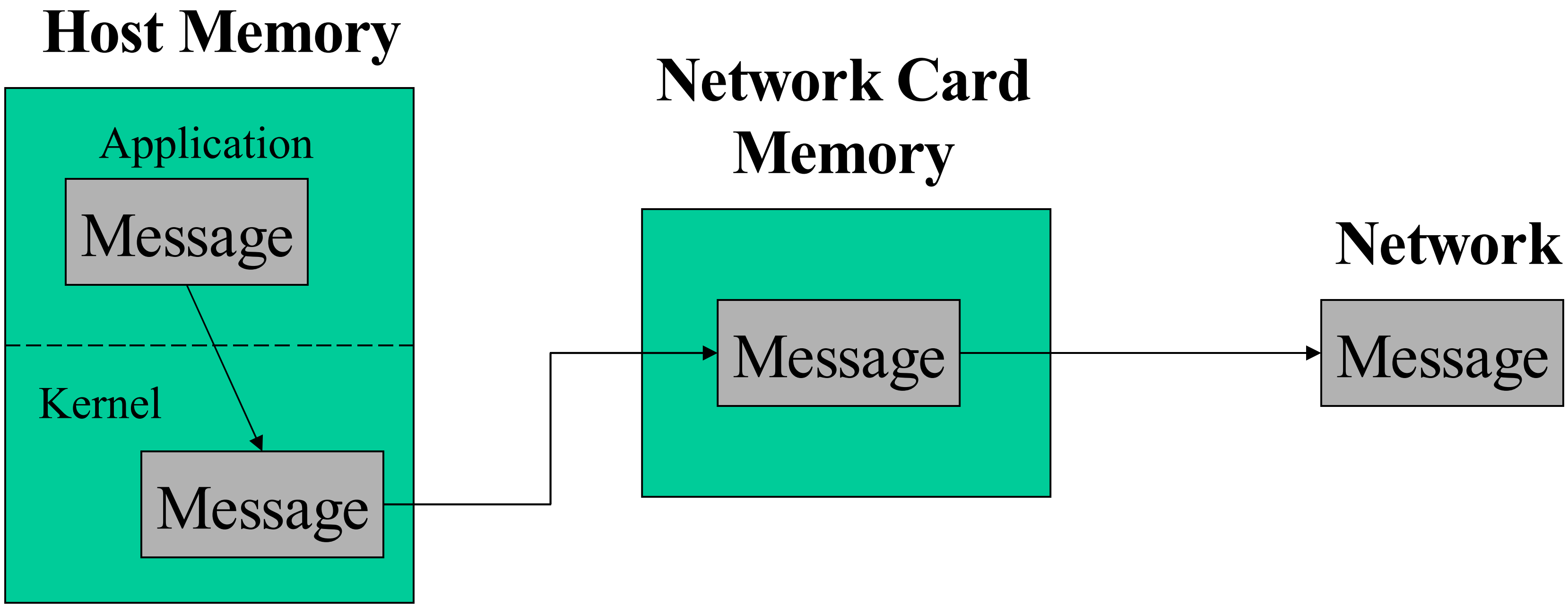
Zero-copy Send

What is it?

- A transmission optimization used on active networks to reduce the delay of sending packets.

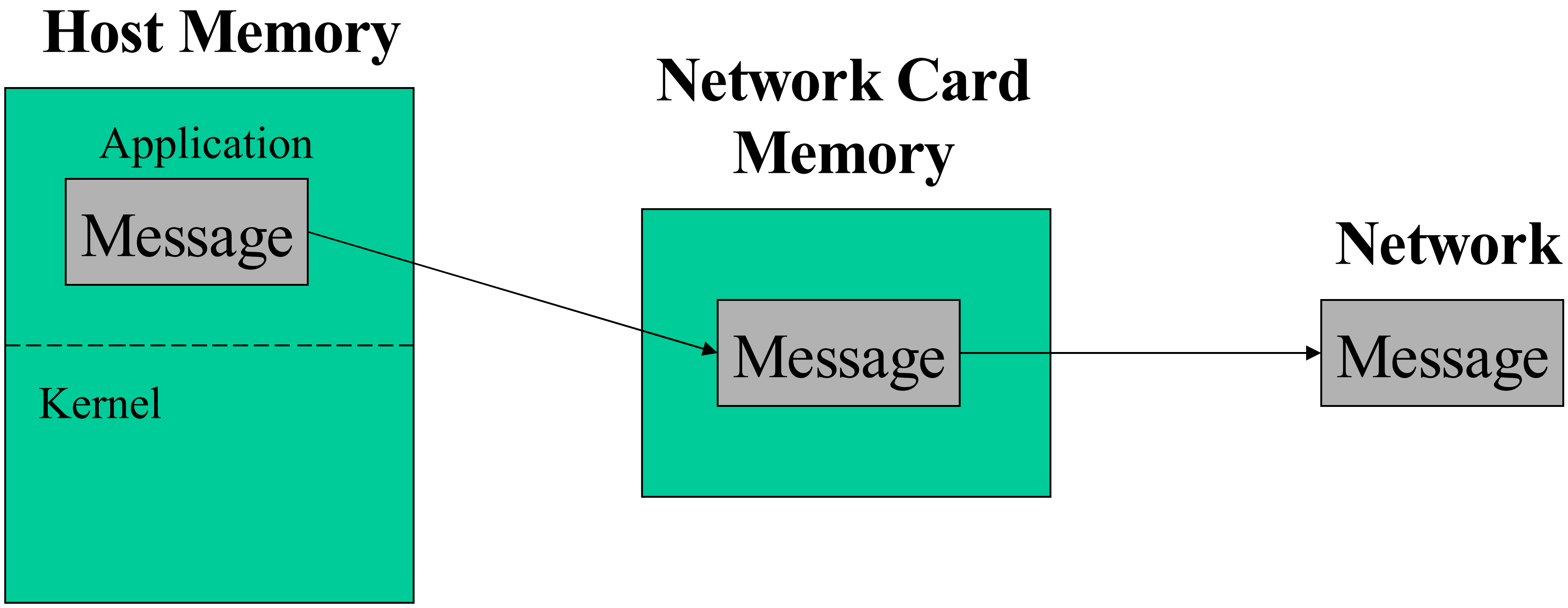
Typical OS behavior

- Application prepares message by writing it in a user-space buffer.
- Application invokes system call
- OS copies message into an internal buffer
- OS pass data to network card (usually by DMA).



Zero-copy Send Behavior

- Application prepares message by writing it in a user-space buffer.
- Application invokes system call (or not, in OS bypass mechanism)
- OS pass data to network card.



Problems

- DMA on PCI might be slower than memcpy because of the synchronization needed to activate the DMA transfer.
- Writing the message directly inside the network card can slow down the application.
- However having first the message being written in an applicaton buffer and then transferred to the network card by DMA increases the latency and the overall overhead of the OS.
- **Typical DMA can be done only by pinning memory (usually makes OS bypass unavailable)**
- Non-snooping caches may create data inconsistencies. But the host can force it.
- Security

Remote DMA

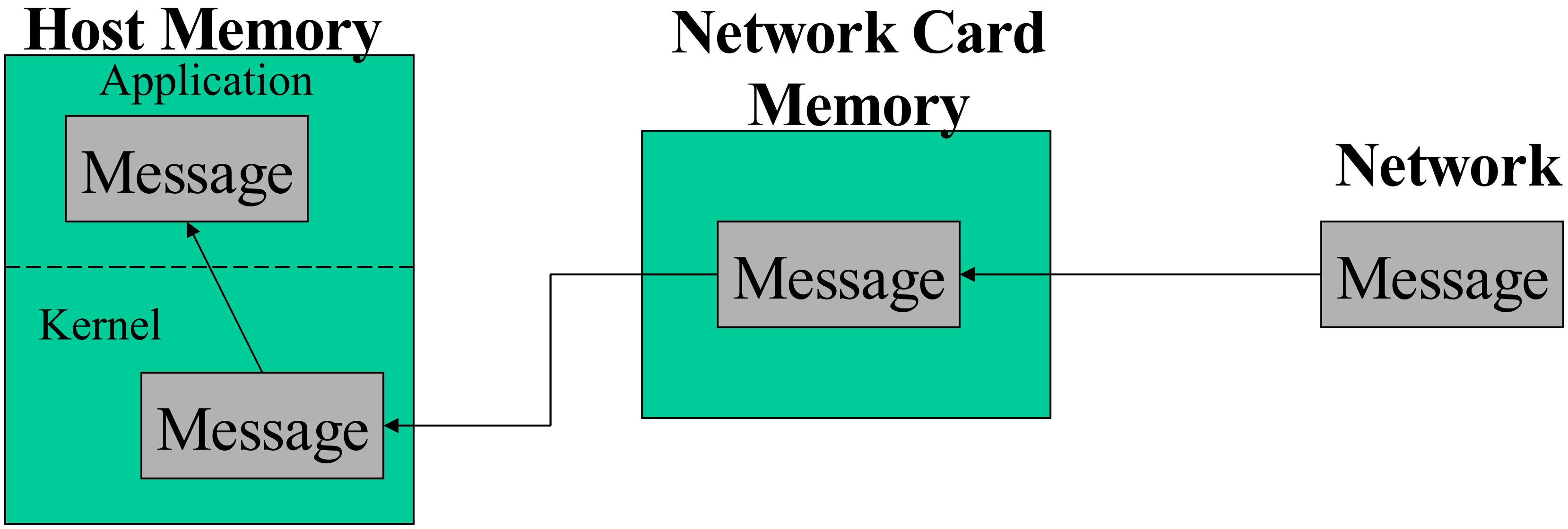
What is it?

- The reciprocal of zero-copy send

  A receiving optimization used on active networks to reduce the delay of sending packets.

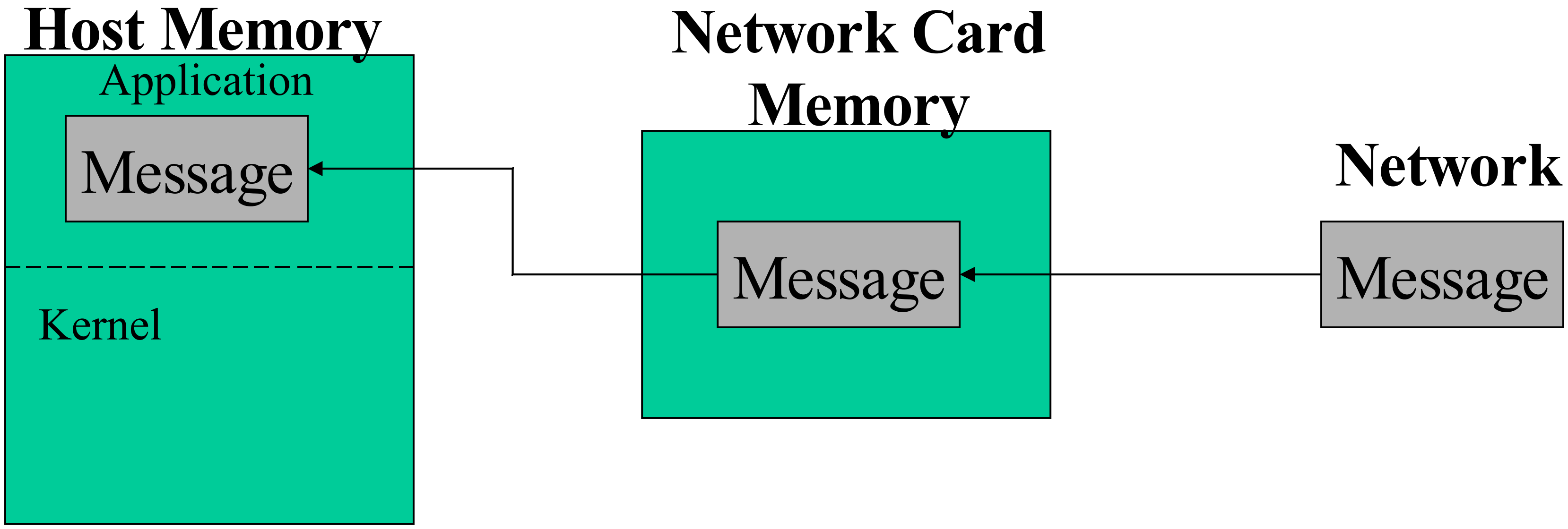
Typical OS behavior

- OS receives message from network card into a kernel buffer.
- Application invokes system call to receive.
- OS copies message from internal buffer into an application level buffer.
- Application reads message from the buffer.



Remote DMA behavior

- Application invokes system call to receive.
- OS copies message from a network card buffer into an application level buffer.
- Application reads message from the buffer.

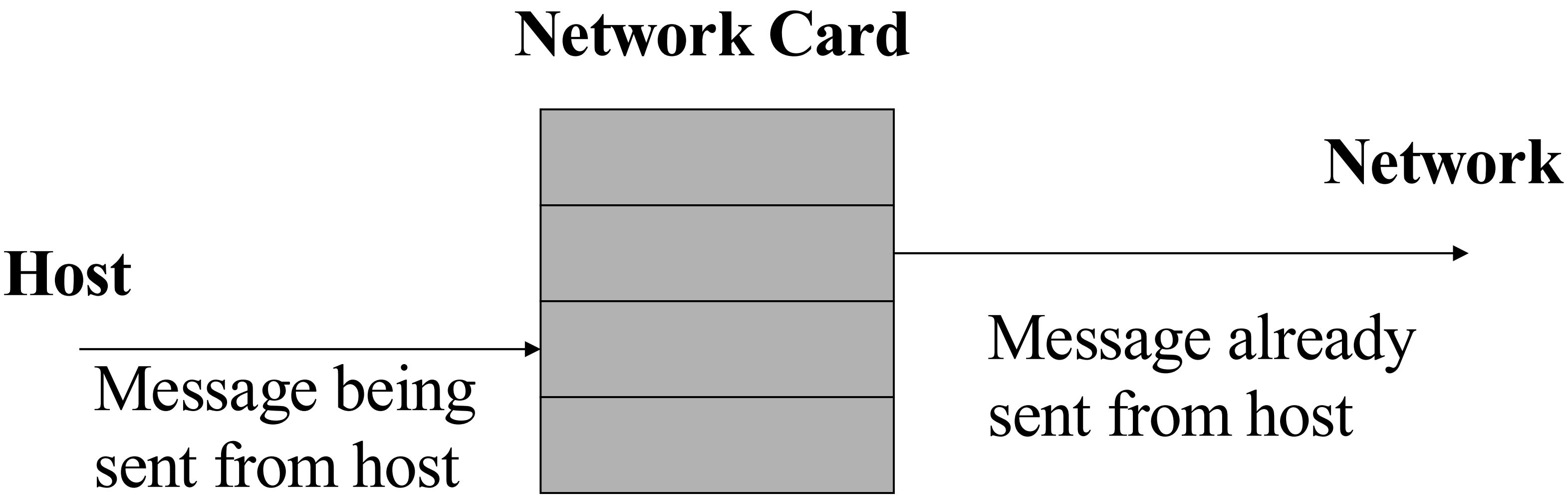


Problems

- The receiving buffer must be known “A priori”.
- Receive call must be invoked BEFORE any receiving starts taking place.
- Receive call doesn’t conform to the typical “read()” paradigm.
- **Very difficult to pin down memory and must be kept up longer.**
- Non-snooping caches may create data inconsistencies. Forcing coherency is harder than in zero-copy send.
- Security

Pipelining technique

- Does not improve latency
- Improves ***dramatically*** the available bandwidth allowing to get close to the network nominal bandwidth
- Can be used in both send and receive operations



Network Card

Network

Host

Message being sent from host

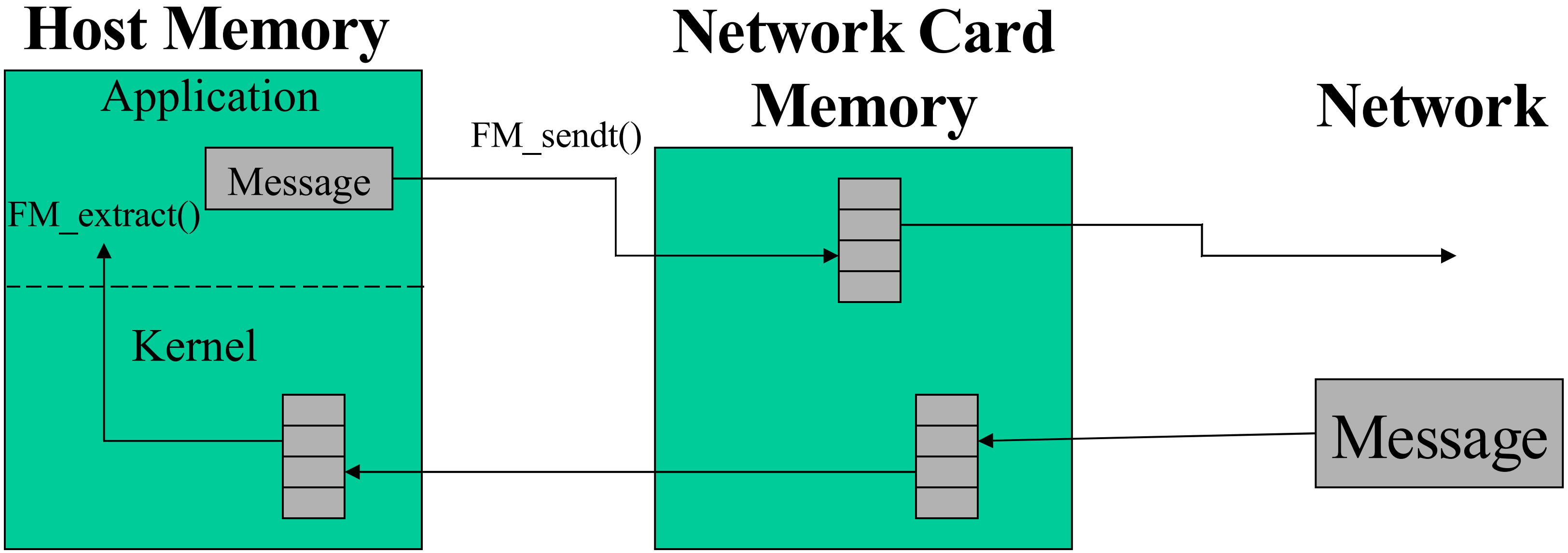
Message already sent from host

GM – official myrinet driver

- NO OS bypass (but streamlined OS access)
- NO zero-copy.
- Remote DMA (only for some architectures)
- Employs the abstraction of “port” in which a single card has different connection ports (a “port” is defined by a specific memory area the network card).

Illinois Fast Messages – Using the Myrinet active networking abilities

- Uses the both zero-copy send and traditional send.
- Pipeline both in send and receive
- OS bypass
- No Remote DMA



FM 1.x Interface

- FM_send(dest, handler, buf, size);
  Traditional send, for long packets.

- FM_send_4 (dest, handler, i0, i1, i2, i3)
  Zero copy send, for small packets.

- FM_extract();
  Receive call.

FM Baseline Myrinet Control Program

(Old Myrinet card)

```
While (1)
   {
   If ((hostsent != lanaisent)  && (SDMA available))
         {
         Send packet;
         Lanaisent++;
         }
   If (packet on Receive)
         {
         Receive Packet
         Activate EBUS DMA;
         }
   If (EBUS DMA completed)
         Notify host;
   }
```

FM *streamed* Myrinet Control Program

- Optimizes peak traffic situations

```
While (1)
    {
    while ((hostsent != lanaisent) && (SDMA available))
        {
        Send packet;
        Lanaisent++;
        }
    while (packet on Receive)
        {
        Receive packet;
        Activate EBUS DMA;
        }
    If (EBUS DMA completed)
        Notify host;
    }
```

FM Myrinet Control Program

(New Myrinet card)

```
While (1)
  {
  If ((hostsent != lanaisent) && (SDMA available))
      Activate SDMA;
  If (packet on Receive)
      Activate RDMA;
  If (SDMA completed)
      lanaisent++;
  If (RDMA completed)
      Activate EBUS DMA;
  If (EBUS DMA completed)
      Notify host;
  }
```

FM 1.x Examples

- Send

```
char buf[500];


FM_send(2, 5, buf, 500);
FM_send_4 (2, 5, 1, 2, 3, 4);
```

- Receive

```
FM_register_handler (5, myhandler )
FM_extract();
```

Full streaming over Myrinet

- Full streaming exists only on upper layers (examples GM-TCP or GM-SOCKS).
- Low level communication based only on packets
- Mandatory for every communication based on a non-virtual circuit paradigm.
- **However higher layer for communication (such as MPI) can dramatically slow down the lower levels (such as FM)**

Problems due to “stream” upper layers

- All data to be transmitted must be into a single contiguous buffer.

Cost of assembling/disassembling messages

- The receiving process decides when to service the network, but is unable to decide how much data to serve

Cost of additional buffer management

FM 2.x

- Low level similar to FM 1.x
- Added flow control at application level.
- Interface changed to reduce interface inefficiencies
  - support receiver flow contol
  - Gather/scatter send

FM 2.x Interface (Send API)

- FM_stream * FM_begin_message (unsigned long dest, unsigned long size, unsigned long handler);
  - Opens a transmission stream with *dest*.
  - Sends up to *size* bytes.
  - The data from this stream will be managed by *handler*.
- void FM_end_message(FM_stream * stream);
  - Close the transmission stream "*stream*".
- void FM_send_piece(FM_stream * stream, void * buf, unsigned long bytes);
  - Sends a message of size *bytes* over a *stream.*

FM 2.x Interface (Receive API)

- int FM_extract(bytes);
  - This is used to verify whether there are messages pending on the network and in that case the appropriate handler is invoked. Bytes is used to manage receive side flow control.
- void FM_receive(FM_stream stream, buf, bytes);
  - This is used inside an an handler to receive an amount of bytes data from *stream*
- unsigned long FM_register_handler (unsigned long id, FM_handler *handler);
  - Register the function *handler* as a stream handler with a certain *id*.
- int handler (FM_stream * stream, unsigned long sender);
  - Prototype of a stream handler.

FM 2.x Examples

- Send

```
FM_stream stream;
Char buf[100], buf2[200];

stream = FM_begin_message (dest, 500, 5);
FM_send_piece (stream, buf1, sizeof(buf1));
FM_send_piece (stream, buf2, sizeof(buf2));
FM_end_message(stream);
```

- Receive

```
FM_register_handler (5, myhandler )
FM_extract(500);
```

- Receive handler

```
char * buf;
int myhandler( FM_stream * str, unisgned long sender)
    {
    buf = malloc(500);
    FM_receive(buf, str, 500);
    Return FM_continue
    }
```

GM – official myrinet driver

- NO OS bypass (but streamlined OS access)
- NO zero-copy.
- Remote DMA (only for some architectures)
- Employs the abstraction of “port” in which a single card has different connection ports (a “port” is defined by a specific memory area the network card).

GM Interface (Send API)

- **GM_ENTRY_POINT** gm_status_t **gm_open (struct gm_port ** p, unsigned int unit, unsigned int port, const char * port_name, enum gm_api_version version);**
  - Creates a new handle *p* on the card *unit* on *port*, assigning a name to the new handle and specifying which version of GM is in use.

- **GM_ENTRY_POINT void gm_send_with_callback (struct gm_port * *p*, void * *message*, unsigned int *size*, gm_size_t *len*, unsigned int *priority*, unsigned int *target_node_id*, unsigned int *target_port_id*,** gm_send_completion_callback_t ***callback*, void * *context*)**
  - Sends a *message* of *size* from port *p* to host *target_node_id* and port *target_port_id* with a certain *priority* and sends a *callback* with parameter *context* when the message is arrived.

GM Interface (Receive API)

- **GM_ENTRY_POINT union gm_recv_event* gm_receive (struct gm_port * *p*)**
  - Receives any event or callback from GM. Return event GM_NO_RECV_EVENT if nothing is pending, GM_RECV_EVENT if a common message is pending or GM_HIGH_RECV_EVENT if a high priority message is pending.

- **GM_ENTRY_POINT void gm_provide_receive_buffer_with_tag (gm_port_t * *p*, void * *ptr*, unsigned *size*, unsigned *priority*, unsigned int *tag*) ;**
  - Allocates a buffer in which GM can receive a message.

- **GM_ENTRY_POINT void gm_unknown (struct gm_port * *p*, union gm_recv_event * *e*) ;**
  - Processes any event or callback returned by *gm_receive()*

Esercizio 1 (A)

Data una scheda myrinet che utilizza un MCP con la seguente struttura:

```
While (1)
    {
    If (hostsent != lanaisent) activate_SDMA();
    If (packet on Receive) activate_RDMA;
    If (SDMA completed) lanaisent++;
    If (RDMA completed) activate_EBUS_DMA();
    If (EBUS DMA completed) notify_host();
    }
```

Esercizio 1 (B)

Tenendo presente che:

2) Il DMA verso l’host attivato dalla funzione activate_EBUS_DMA() usa il quarto canale di priorita’ (canale 3).
3) La struttura di un pacchetto ricevuto consista di un header di 32 byte, una prima porzione di 256 byte, e una seconda porzione di altri 1024 byte,
4) I primi due canali a priorità più alta (canali 0 e 1) siano inutilizzati
5) I pacchetti in ricezione vengano ricevuti in pipeline in due buffer alternativi, all’indirizzo LANai 0x2000 e 0x4000.

Esercizio 1 (C)

La criticità dell'informazione contenuta nella prima porzione del payload suggerisce di far si che il MCP trasferisca questa informazione all'host alla massima priorita' possibile. Si richiede quindi che l'informazione critica (e solo essa) dovra' essere trasferita all'indirizzo fisico dell'host 0x50000 sfruttando l'opportuno canale e successivamente l'MCP dovrà segnalare all'host il completamento del trasferimento tramite una interruzione. Si noti che la procedura di ricezione vera e propria del messaggio non deve subire alterazioni.

Si descriva:

D) come va modificata la struttura del MCP per ottenere l'obiettivo descritto sopra.

E) Il codice di tali modifiche.

F) Si discuta l'opportunita' di tale modifica,e in quali casi essa risulta giustificata.

Esercizio 2 (A)

- Si supponga di avere un cluster di macchine interconnesse da reti attive. Su ognuna delle macchine si trova una applicazione che per accelerare la comunicazione con le altre utilizza gli Illinois Fast Messages 2.x. Ognuna di queste applicazioni in vari momenti temporali può aver bisogno di interrogare tutte le altre macchine sul loro stato attuale, e di effettuare un calcolo su di esse una volta ricevute tutte. Tuttavia tra la richiesta delle informazioni e la loro ricezione l’applicazione deve essere libera di eseguire altri calcoli e in particolar modo di processare altri messaggi.

Esercizio 2 (B)

- Date le seguenti informazioni:
  - 1) La dimensione dello stato e’ fissa e costituita da 4 byte.
  - 2) Il calcolo sui diversi stati viene effettuato da una funzione di libreria *void calcolo_sugli_stati(struct _stato * stato, int numero_stati)* dove *stato* e’ un puntatore a una lista di stati e *numero_stati* rappresenta il numero di elementi in questa lista.

- si implementi la funzione start_computation() che attiva la computazione, nonchè gli handler necessari a gestire la comunicazione (in entrambi i sensi).