FORTH-ICS / TR-123 May 1994

Telegraphos:

High-Speed Communication Architecture for Parallel and Distributed Computer Systems

Manolis Katevenis

ABSTRACT: *Telegraphos* is an R&D project in computer communication. It uses hardware switches for building high-speed multiprocessor or local area networks; preventive flow control eliminates packet dropping, and dedicated buffers per VC offer congestion tolerance. The network interfaces have low complexity because they never need to retransmit packets, and because they use a single address space architecture. Message passing is done with the *remote write* primitive; overhead is minimized because address translation also performs message protection. Other remote memory operations, including eager updates and hardware monitors, provide effective and low-cost support for virtual shared memory. *Telegraphos I,* our first prototype, is currently being built.

Telegraphos:

High-Speed Communication Architecture for Parallel and Distributed Computer Systems

Manolis Katevenis

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Science and Technology Park, Heraklio, Crete
POBox 1385, GR-711-10 Greece
E-mail: katevenis@ics.forth.gr, Tel: +30 (81) 391664, Fax: +30 (81) 391661

Technical Report FORTH-ICS / TR-123 – May 1994

Work performed under ESPRIT Contract 6253 "SHIPS"

ABSTRACT: *Telegraphos* is an R&D project in computer communication. It uses hardware switches for building high-speed multiprocessor or local area networks; preventive flow control eliminates packet dropping, and dedicated buffers per VC offer congestion tolerance. The network interfaces have low complexity because they never need to retransmit packets, and because they use a single address space architecture. Message passing is done with the *remote write* primitive; overhead is minimized because address translation also performs message protection. Other remote memory operations, including eager updates and hardware monitors, provide effective and low-cost support for virtual shared memory. *Telegraphos I,* our first prototype, is currently being built.

KEYWORDS: *high-speed network, high-speed switch, congestion tolerance, dedicated buffer per VC, high-speed network interface, Telegraphos, remote write, eager updating, VSM hardware support, message passing in single address space.*

This text is available in Postscript form, by ftp, from server "ftp.ics.forth.gr"
Login as "anonymous", give your e-mail address as password
Directory: "tech-reports/1994"
Files: "94.TR123.Telegraphos.README", "94.TR123.Telegraphos.ps.Z"

Source Directory: ~kateveni/telegraphos/tr123

CONTENTS

1. Overview of the Telegraphos Project

Three key hardware ingredients are needed to make high performance parallel and distributed computer systems:

- high-speed computation,
- → high-speed communication, and
- high-speed peripheral (I/O) devices.

Telegraphos is a research and development project dealing with the second of these three components. The name of the project comes from the Greek words τηλέ, meaning *remote*, and γράφω, meaning *write*, because the *remote memory write operation* plays a central role in this system.

Contemporary parallel and distributed computer systems consist of processors, each with a local memory hierarchy and possibly peripheral devices, interconnected by a high-speed network. The network forms the communication part of the system. There are two key components in making this network fast:

- → high-speed links and switches, and
- → high-speed network-to-processor (memory) interfaces.

The Telegraphos project deals with both of these sub-systems. Section 2 describes the Telegraphos network, and explains why we use point-to-point links and a back-pressure flow-control protocol over them, which guarantees that no packet will ever be dropped. That section also describes the dedicated buffer architecture of our switches, explains why this architecture offers congestion tolerance, and argues why this is important in parallel and distributed systems.

The Telegraphos network-to-processor interface architecture is presented in sections 3 and 4. Section 3 focuses on the remote-write operation, and shows how this allows low-overhead message passing, and how it saves the complexities of receive buffer overflows and message reassembly from multiple packets. Section 4 describes Telegraphos' support for the shared memory model of parallel computation. We investigate mechanisms that are less expensive and less complicated than directory-based cache coherence: Telegraphos includes a number of hardware primitives to support virtual shared memory. Besides remote writes, we provide remote atomic operations, and blocking and non-blocking remote reads. Additionally, Telegraphos I provides *eager sharing:* writes into a page are automatically reflected, in hardware, into all copies of the page that may exist on multiple nodes. Thus, for each page, the compiler, the run-time environment, or the OS can choose to copy it locally or to access it remotely; in case of local replication, a choice of coherence protocol is provided between write-invalidate and write-update. These choices – static or dynamic – are assisted by hardware counters, counting the number of local and remote accesses per sharable page.

Finally, section 5 presents our current and future R&D activities. On the research side, we are working on high-speed link and switch technology, merging and diverging virtual paths, static versus dynamic routing, error recovery and synchronization methods, hardware support for fast message passing and for virtual shared memory (possibly leading towards an integration of the shared-memory and message-passing models), and on thread and data placement algorithms. The Telegraphos project also has a strong development component: we are building both the hardware and the systems software of the system.

Telegraphos prototypes connect workstations within an office space through a network that is so fast as to make the system into something more than a workstation farm: a true parallel computer. *Telegraphos I,* our first prototype, is currently in the last phases of its hardware design, and will be built soon. It will interconnect DEC Alpha [Sites93] workstations, by plugging into their TurboChannel I/O bus. Emphasis is on rapid prototyping, so the technology used is conservative: FPGA's, SRAM's, and TTL buffers; clock frequency will be about 25 MHz, and link throughput will be about 200 Mbits/s in each direction. A 4-by-4 switch will occupy one board, and a processor-network interface will also occupy one board. Work on *Telegraphos II* is starting now. CMOS standard-cell ASIC technology will be used to reduce the size and increase the speed of Telegraphos I. Links will run in the range of 500 Mbits/s, and a 4-by-4 switch will be integrated in a single chip; another chip will implement the processor-network interface. For other future Telegraphos versions, we are considering designing key sub-systems in full-custom CMOS, and using low-voltage-swing signals between chips, in order to achieve much higher performance. For systems software, we are modifying the *Mach* parallel operating system, in order to make it the OS of Telegraphos.

1.1 Underlying Principles

The application of RISC [Kate85] principles to modern processor architecture has yielded high-speed computation engines [HePa90]. Our current work in developing hardware building blocks for high-speed communication is inspired by those same principles:

- *"Hardware should provide primitives – not solutions"* [HePa90 – pp. 121, 124]. Telegraphos provides the remote write operation, which is as primitive as the *store* instruction in RISC processors; building various message passing and other information transfer services on top of it is left to the software. Telegraphos also provides remote reads (blocking and non-blocking), remote atomic operations, write multicasting, and hardware page access counters; making decisions about data placement and replication, and using these primitives to efficiently implement virtual shared memory is left to the software.
- *"Support the most frequent case in hardware* (and do *not* spend precious hardware to support the less frequent cases)." In Telegraphos, all per-packet processing is done purely in hardware, both in the network switches and in the processor-network interface. Our packet time (the inverse of the packet rate) is only a fraction of a microsecond, and it will decrease towards a few tens of nanoseconds in future versions; as a consequence, there is no room for software in this area. However, nothing at any higher level is done in hardware. In particular, no error recovery and no packet retransmission is provided in hardware (preventive flow-control excludes the case of massive packet loss due to congestion, thus only leaving the infrequent cases of packet loss due to hardware failure or excessive electrical noise).
- *"Simplify (support regularity)."* Examples of this can be found throughout the detailed hardware design. One particular case is the packet size, which is fixed, and the packet format, which is adjusted so as to minimize field shifting and multiplexing.

One more principle plays a central role in the Telegraphos architecture, although it is not one of the "RISC family" of principles: *"Communication is more expensive than storage and computation."* To see why this is true, start with electrical communication, and consider first the on-chip case. One average "non-local" wire, of length 5 mm on a 1μm-technology chip, consumes a silicon area of about 20,000 μm^2; in a CMOS SRAM, approximately 40 bits of information can be stored in a similar area. Or else, in a similar technology, one 32-bit RISC integer processor can fit in as few as 5 mm^2, i.e. the same area as that occupied by just 250 of the above wires. As technology progresses, memory and logic area shrinks faster than wire area (for wires that traverse a fixed percentage of the chip length), so the above argument becomes even stronger. Second, consider the off-chip case. The chip periphery constitutes a communications bottleneck: only a few hundred pins are available for the over one million transistors inside the chip to communicate with the rest of the world; additionally, these pins have to operate at lower rate than the rest of the chip, in order for power consumption not to be excessive.

Then, consider communication through optical fibers. As we will see in section 2.2, when using preventive flow control, the necessary buffer space is related to channel throughput times propagation latency. Take a typical approximate cost of 10 KECU per kilometer for a cable containing 10 optical fibers, excluding terminators, optical transceivers, and electro-optical interfaces. Assume that each fiber carries approximately 25 Gbits/s of traffic. For each such fiber, the throughput times latency per meter of length is roughly 25 Gb/s $\times$ 4 ns, or 100 bits, and the cost is 1 ECU. On the other hand, at a typical DRAM cost of 32 ECU per MByte, or at a typical SRAM cost of 320 ECU per MByte, one can buy approximately 25 to 250 Kbits for 1 ECU. We see that communication (optical fiber) is about 3 orders of magnitude (10^3) more expensive than storage (DRAM, SRAM), when comparing memory space against throughput-latency product.

In Telegraphos, the observation that communication is more expensive than storage and computation led us, firstly, to back-pressure (preventive) flow control, so that throughput is not wasted by transmitting packets that would subsequently be dropped and have to be retransmitted, and, secondly, to a switch architecture with dedicated buffers per virtual path, which, as explained in section 2, spends more of the less expensive resource – buffer space and flow-control circuitry – in order to fully utilize the scarce resource: link throughput.

2. Switch Architecture – Congestion Tolerance

Some of the network architectures have their roots in choices that were made many years ago, for technologies very different from the those of today. Some network designs are still under the influence of how telephony networks used to be many decades ago. Some other designs follow the style of microcomputer buses. A popular set of assumptions, which however should be questioned today, is about low-throughput, high-error-rate links, and network sources of highly controllable or predictable nature. Telegraphos strives to abandon such influences from the past when they are not justified by the technology of the present or by the intended applications. The need for such new network architectures has already been pointed out – see for example [Kung92].

2.1 *Point-to-Point, Multi-Wire Network Links*

Multiple-access physical media – e.g. wires driven by multiple (tristate) drivers – are not practical anymore for high-speed networks. One reason is the underutilization of the medium resulting from the driver turn-around delay (the delay from shutting off one driver to turning on another): as the packet time becomes shorter, because transmission speed increases, this turn-around delay becomes an increasing portion of the total time. A second reason is that arbitrating among physically separated sources is complex and slow.

This observation has led many parallel machines to use point-to-point links: links with a single driver and a single receiver on each of them. Full duplex communication is provided by having a separate link in each direction. The single driver eliminates the turn-around and arbitration problems; the single receiver ensures that receiver circuitry and wires are not underutilized, as they would be if they carried information destined to one of many receivers. Nowadays, distributed systems, and even bus-based multiprocessors follow in the direction of parallel machines. For example, Ethernet gives its place to rings like FDDI, or even to ATM; and backplane buses are likely to give their place to networks like SCI [SCI92].

For the above reasons, Telegraphos uses point-to-point links. A network with point-to-point links must be a ring, or it must be fully connected, or it must use switches. Rings have severely limited throughput; fully connected is impractical for large systems; so Telegraphos uses switches. Each Telegraphos link consists of multiple wires, carrying multiple bits in parallel. The specific width of the link is technology dependent; anyway, it is less than the packet size, so it takes multiple clock cycles for a packet to be transmitted. We use electrical transmission over wires, rather than optical transmission over fibers, because the former is considerably less expensive for short distances (electro-optical interfaces at Gb/s speeds are very expensive), and because the propagation delay skew among the parallel wires can be controlled to quite low values for links up to a few tens of meters in length.

2.2 Back-Pressure Flow Control: Never Dropping Packets

Some network transmission physical media have (or had) a relatively high error rate. In such networks, packets are lost so frequently that recovering them has to be considered as one of the normal and frequently performed functions of the network. These networks have to use acknowledgement, time-out, and retransmission protocols in order to cope with lost packets. If this were the case in a high-speed network, with links in the Gbit/s range and with bit error rate in the 10^{-4} range, then these protocols would have to operate at a rate of one retransmission every few microseconds. This would probably necessitate the implementation of these acknowledgement, time-out, and retransmission protocols in hardware, leading to considerable complexity and cost.

Fortunately, the situation is much better for carefully designed short distance electrical links with good shielding: the error rate is extremely low. For example, the "ATOMIC" project team at USC/ISI has seen *not a single error* during their 2,000 hours of validation testing (10^{15} bits of data transmitted) for multi-hop paths through "Mosaic" switches connected by short-distance ribbon cables [CFFS93 – section 2e]. Under these circumstances, recovering lost packets is such an infrequent operation that it should be left to software – no time-out and retransmission hardware is needed. Telegraphos does precisely that: the hardware provides error detection, while error correction is the responsibility of software. We are currently working on various algorithms for error recovery: see section 5.3.

In traditional networks with high error rate, once there is in place a protocol for retransmission of lost packets, this can also be used for packets that are dropped due to contention. This has made flow (rate) control protocols or congestion control mechanisms popular, that on the average achieve a relatively low rate of packet dropping, but do not totally prevent packets from being discarded and retransmitted. However, in our case, in high-speed data networks with reliable links, such flow control mechanisms would be unacceptable, because they would re-introduce the need for hardware-implemented acknowledgement, time-out, and retransmission protocols. Instead of that, Telegraphos uses window-based *preventive* flow control at the individual link level ("back-pressure"), which guarantees that packets will *never* have to be dropped; this mechanism was also used in our previous designs [Kate87 – section II.A]. This flow control mechanism is much simpler to implement in hardware than time-out and retransmission protocols, and it also leads to better utilization of network throughput. Several modern switches use similar flow control – see for example [CFFS93], [Dunb93], [C104].

Figure 2.1 illustrates the preventive flow-control that we use. Transmission of a packet along a link – from *switch 0* to *switch 1* – is only allowed when *switch 0* knows that *switch 1* is guaranteed to have enough buffer space for the reception of the packet. To accomplish that, *switch 0* maintains a count of available *tickets,* representing permissions to send packets (reservations of buffer space). Every departing packet decrements the count; transmission is only allowed when this count is non-zero. At the other end, whenever new buffer space becomes available, a flow-control ticket is sent back. As long as we guarantee that

$$\text{(Buffer Space)} \geq \text{(Ticket-Packet Round-Trip Delay)} \times \text{(Peak Throughput)}$$

no performance degradation results from this back-pressure scheme. In this simple case, the peak throughput quoted above is the maximum link capacity. In section 2.4, when buffer space is reserved individually for each virtual circuit (VC), that space is computed using the maximum instantaneous throughput of the particular VC that the network designer or manager wants to

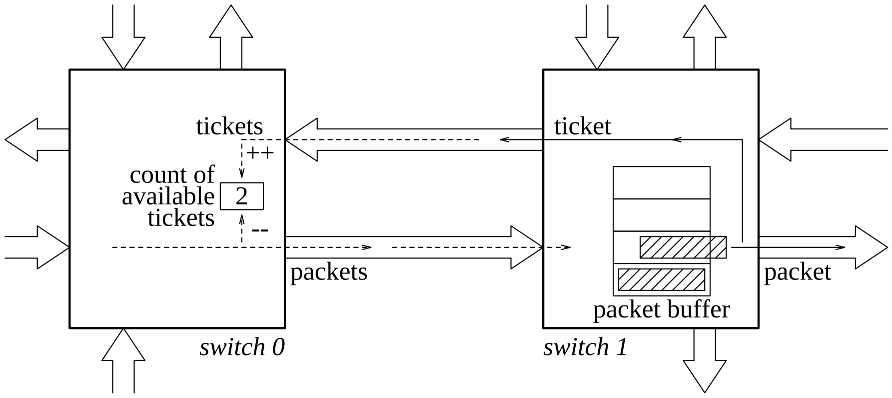

Figure 2.1: Ticket (window) based back-pressure flow-control at the link level 5

allow (in Telegraphos I, any single VC is allowed to reach the maximum link capacity). The round-trip delay in the above equation is the worst-case time from the departure of a packet from *switch 1*, till the corresponding ticket reaches *switch 0* (assumed to have no tickets available prior to that time), causes another packet to be sent from *switch 0* to *switch 1*, and this latter packet reaches *switch 1* and subsequently departs from there (assuming it was the head packet in the buffer when it arrived, i.e. the buffer was empty just prior to its arrival). As we commented in section 1.1, the cost of the above buffer space is several orders of magnitude smaller than the cost of the corresponding communications medium. To put the above equation in perspective, let us note that in networks without preventive flow control, buffer space requirements are not related to throughput or delay, but they depend on the network load as a percentage of the network capacity, the burst size, and the desired upper bound for the packet loss probability; these requirements grow infinitely when the network load approaches the network capacity (for fixed packet loss probability).

2.3 Load Unpredictability and Congestion Tolerance

Traditional networks are usually designed so as to operate with a reasonable efficiency when the traffic load presented to the network by its sources does not exceed a certain limit – often this limit ranges around 50 to 90 percent of the maximum network throughput capacity. If the network load exceeds this limit, then a phenomenon usually described as *throughput collapse* occurs: even though the sources request the network to deliver an increasing amount of traffic, the network actually delivers a decreasing amount of traffic, and the delay suffered by this traffic increases dramatically. The word *congestion* is also used in this context, sometimes as a synonym of throughput collapse. In this paper, however, we will reserve this word – congestion – to mean the state in which the traffic load presented to the network by its sources approaches or exceeds the maximum network throughput capacity. In traditional networks, congestion leads to throughput collapse, but *not* so in Telegraphos, as explained below.

The reason why traditional networks can operate successfully with such a "vulnerable" design is *statistical multiplexing* and the predictability (or controllability) of their sources. The average throughput requirements of each source are low compared to the total network throughput, and the momentary peak throughput requirements multiplied by their expected duration – the burst sizes – are low compared to the total buffer space provided in the network. Since *many, independent* such sources are multiplexed on the network, the laws of probability can be taken advantage of in order for the occurrence of congestion to be made as unlikely as the designer and the operator (manager) of the network want it to be: by controlling the number and the nature of the sources to which access to the network is granted at any given time, the probability of congestion can be made as small as desired.

Unfortunately, none of the above good properties seems to be true for parallel computer networks [Kung92, p. 82]. Data traffic in a multiprocessor can often be *highly bursty:* there are times when processors want to send long messages, using by themselves as much of the network throughput as they can get; and there are times in the parallel programs and places (processors) in the network where "hot spots" arise. For example, in a study of eight scientific applications explicitly written for message-passing multicomputers, [CHKM93] found a large variation in message lengths. For half of the applications, the average message length was 8 Kilobytes or larger. One of the applications had an *average* message length of half a *Mega*byte! When such a long message is sent, we want it to be sent as fast as possible; usually, this means that we want this single message to saturate, if possible, the network links through which it passes. Obviously, the burst sizes are no longer small compared to the buffer space provided in the network. Also, the sources multiplexed on the network are not necessarily so numerous for the laws of large numbers of probability to apply, since it is desirable for a single source to be able to saturate the links through which its messages pass. Additionally, there is no reason to believe that the network sources are independent of each other, since usually most of them are threads of the same parallel program, often operating in synchrony: hot spots, e.g. near synchronization points, are quite frequent. Neither is it reasonable to apply admission control – to deny access to the network by some sources in order for the rest of them to enjoy a guaranteed level of service: all sources are "equal" partners of a parallel program, and they must all make forward progress in order for computation to proceed smoothly.

In short, it is *un*desirable and *in*feasible to ask for parallel programs or distributed applications (e.g. multimedia) to be *predictable* (statistically simple to describe) sources of network traffic: this would over-constrain the development of parallel algorithms, and it would make the writing of parallel programs exceedingly difficult. Neither is it possible to make parallel programs *controllable* sources of network traffic: end-to-end flow control mechanisms have such slow response that the overall performance of the parallel program would suffer unacceptably. It follows that parallel and distributed computer networks need *congestion tolerance* – a property that had attracted little attention in the past. By this term we mean that the network should behave robustly and smoothly in the presence of an increasing load of requests, which reaches and considerably exceeds the network capacity for long periods of time. Excess traffic should be made to wait at the sources, rather than entering the network and thus creating problems. No throughput collapse should occur, because if that were allowed, the system could often fall in that state and its performance would be seriously impaired.

Congestion tolerance is important not only for systems running parallel programs – it is also important for high-speed distributed computer systems, e.g. *gigabit local area networks.* In addition to the above problems, such networks have to cope with large mismatches in throughput [Kung92, p. 82]: some sources on the network – e.g. high-speed servers or supercomputers – can send very long messages to destinations that are not capable of receiving them at such a high rate – small workstations or bridges to slow networks like Ethernet. If proper action is not taken to throttle the high-speed source, the network will be swamped with data. Again, old solutions are no longer applicable [Kung92, p. 83]. We cannot "over-design" the network so that it never gets congested, since that would require enormous network throughput. We cannot apply admission control, since that would allow the network to be monopolized by a few heavy-load connections. We cannot effectively apply end-to-end flow control, because that responds too slowly to network load fluctuations. We cannot restrict the peak throughput of sources to a small multiple of their average throughput, since the latter is usually minuscule while we want the former to be very large. And we don't want the network to provide very large buffering inside itself, because that merely postpones the solution of the throughput mismatch problem by carrying it over from the sources to the network itself.

2.4 Congestion Tolerance: Dedicated Buffers per VC

There are two basic reasons why throughput collapse occurs in many traditional networks under congestion: packet dropping/retransmission, and head-of-line blocking. Let us consider packet dropping and retransmission first. Under congestion, the network buffers are mostly full, because they are filled faster than they get emptied. If there is no flow control to stop the transmission of packets under these circumstances, then packets that arrive at nodes where buffers are full have to be dropped. This means that part of the network traffic is *wasted,* since packets that are dropped will have to be retransmitted, in one way or another, at a later time (the property of telephony, where bit dropping results in quality degradation but not in retransmission, does not apply here). Under congestion circumstances, packet dropping and retransmission may become a regenerative phenomenon, leading to instability and throughput collapse: retransmissions increase the network load, causing stronger congestion and more packet dropping, which triggers more retransmissions, and so forth. As explained in section 2.2, Telegraphos uses preventive flow control so as to never drop packets, and thus this cause of throughput collapse does not apply in our case.

The second cause of throughput collapse is *head-of-line blocking,* which is particularly prevalent when the packet flow is stopped in order for packets not to be dropped. Head-of-line blocking is especially pronounced and easy to explain in *input queueing,* as illustrated in figure 2.2*(a).* Input queueing is the switch organization where packets entering through an incoming link are stored in a *FIFO* queue, forming a "waiting line". When the packet at the head of this line ("head-of-line" packet) can be transmitted over its desired outgoing link, then this packet is withdrawn from the input queue, and the next packet on this queue is examined. If however the head-of-line packet cannot proceed, e.g. because its desired output is busy at the moment, as in figure 2.2*(a)*, then all packets waiting behind it are also blocked (hence the term "head-of-line blocking"), even if *their* output is not busy. As an analogy think of a narrow street where cars that want to turn left, but have to wait for gaps in the opposite-going traffic, block cars that are

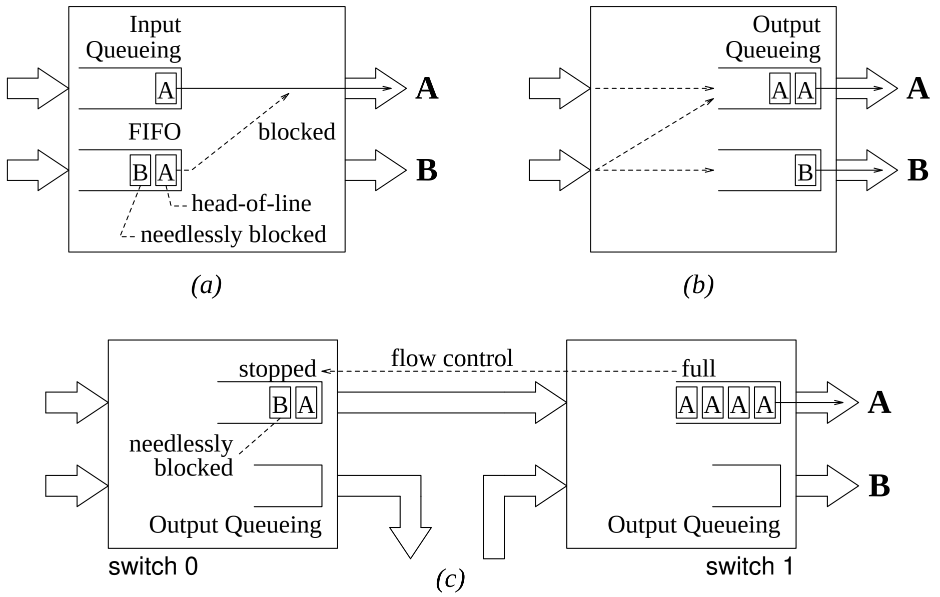

Figure 2.2: Head-of-line blocking: input queueing (a), output queueing (b), multistage network (c)

behind them and want to go straight. Because of this blocking property, this scheme has a very poor performance: even under the ideal assumptions of independently distributed random packet destinations (no bursts) and large buffer space (queues can hold many packets), the switch saturates when the link utilization reaches approximately 60% [HlKa88, p. 1590].

The individual switch performance can be improved relative to input queueing, e.g. by going to *output queueing,* as shown in figure 2.2*(b)*. Here, the FIFO memories must support a larger (peak) write-throughput, but no head-of-line blocking occurs at this level. However, there is some similarity between an output queue in one switch and an input queue in the next switch downstream along the corresponding link. Figure 2.2*(c)* illustrates precisely that: in some sense, the top queue in switch 0 is just an extension of the queues of switch 1; flow control arranges that the former is stopped whenever one of the latter is full. Because this extension is not situated inside switch 1, the head-of-line blocking phenomenon still occurs, even though the switches are organized using output queueing, as the figure illustrates. This situation is especially characteristic of *wormhole routing* [DaSe87], [C104]. In wormhole routing, the queues in the switches have a capacity of only a few *flits* – flow control digits – while a packet usually consists of many flits; thus, a packet will usually be spread among multiple switches. If the head of a packet becomes blocked because of a collision with another packet in some switch, then many queues and switches, upstream from the place of collision, also become blocked, thus leading to poor performance. For example, in [Dally90] it was found that the network saturates at about 20 to 30 percent of its capacity when wormhole routing (1 lane in figure 8) is used, with packet size just 25% larger than the buffer size. Notice the similarity of the flit-packet situation with a network that uses packet-level flow control, and which has buffers that are large enough to hold a few packets each, but where traffic is highly bursty, so that multiple packets with the same destination follow closely after one another.

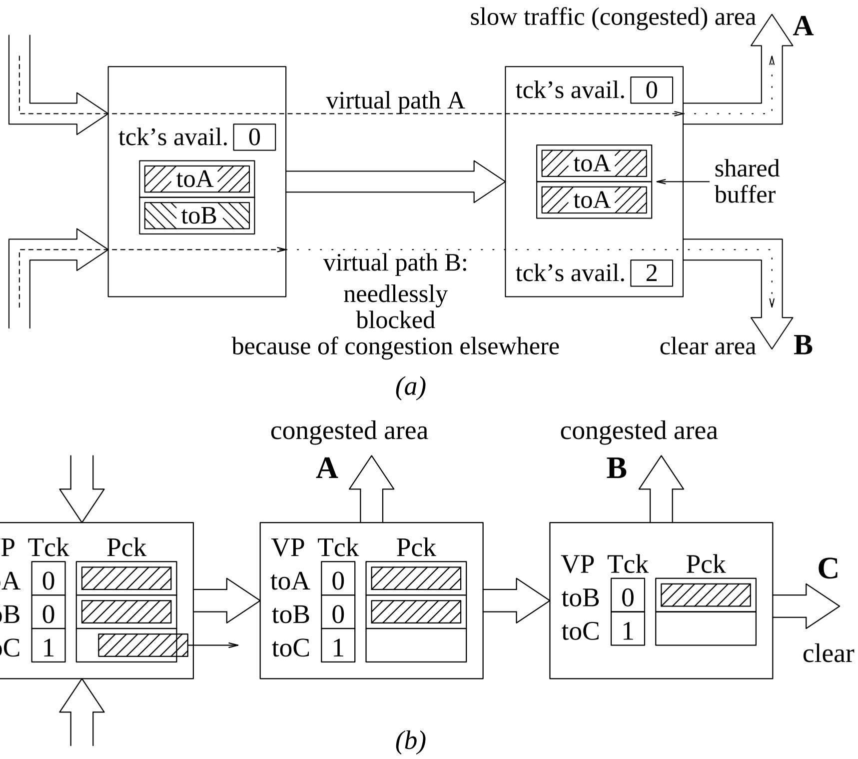

Figure 2.3: Buffer sharing under congestion (a); dedicated buffers and congestion tolerance (b)

Figure 2.2 gives the impression that it is the *FIFO* discipline of buffering inside the switches which is responsible for the phenomenon of head-of-line blocking. This is true inside a single switch, while it is only partly true for multiple levels of switches. The other half of the story is the uncontrolled *sharing of buffer space* among the different flows. Buffer sharing, coupled with incomplete (coarse grain) flow control information, creates a relationship equivalent to a single FIFO queue (input queue) among the buffers in upstream switches and the buffers in downstream switches. Figure 2.3*(a)* illustrates this situation. In this figure, we assume that all packets inside a switch are candidates for transmission at any time – no FIFO order is imposed among packets with different (final) destinations. Area *A* happens to be heavily loaded, while area *B* is currently lightly loaded. Unfortunately, the shared buffer in the second switch happens to be occupied by packets that all wait to go to *A* – something quite likely to happen in the presence of bursty traffic. Because of the slow traffic in area *A*, there are currently no flow-control tickets for these packets. Under these circumstances, all links in this figure needlessly remain idle, while the packet that wants to go to *B* could be transmitted if it were not blocked by the lack of buffer space in the second switch.

Head-of-line blocking, in any one of the forms under which it may appear, results in inefficient use of the link throughput. As we argued in section 1, link throughput is the scarce resource in networks. To solve this problem, Telegraphos uses dedicated buffers per virtual circuit (VC) in each switch, and implements flow-control at the granularity of virtual circuits. This is equivalent to maintaining *separate* (output) queues in switch 0 of figure 2.2*(c)* for the packets destined to *A* and the packets destined to *B* (as well as for all other destinations in the network). Thus, Telegraphos achieves *congestion tolerance:* network utilization increases monotonically to 100% as the load offered to the network increases up to and beyond the network capacity – simply, the sources are throttled, and the network is utilized at its peak capacity. Dedicated buffers also provide deadlock-free routing, irrespective of the underlying network topology.

Figure 2.3*(b)* illustrates how the Telegraphos switches operate. In this figure, the VC's going to A and B have their buffers full and no tickets available, because A and B lead to areas of heavy traffic. However, virtual circuit C, which goes to a clear area, has its own, dedicated buffers, which are free; since tickets are available for C, a packet with that destination (lower left) can proceed. We have been working on switches with this architecture since several years ago: see [Kate87], [KaSC91]; similar architectures were also used in "TYMNET" [Tyme81, p. 395], considered in an early "TRANSPAC" design [SiDa79], and studied in [Dally90]. The buffer space to be dedicated to each VC should be no less than the ticket-packet round-trip delay times the maximum instantaneous rate to be allocated to this VC. In Telegraphos I, we let this peak rate per VC be equal to the peak rate of an entire *link* (the round-trip delay is one packet time, so the buffer space per VC is one packet). Thus, any burst of packets in any single VC can momentarily use the totality of the link throughput that remains unused by the other VC's on the link, up to the total link throughput (without any delay for throughput requests and throughput grants, as would be the case for end-to-end flow control). In large networks, with many VC's, it would be quite expensive to dedicate the above buffer space to each and every VC; we are currently working on architectures that address this problem by grouping together virtual circuits into virtual paths.

2.5 Telegraphos I Switch Block Diagram

Telegraphos I is our first prototype of a Telegraphos system. It was designed with rapid prototyping in mind, in order to provide a first validation and test vehicle for our architecture. Thus, Telegraphos I uses only commercially available components – mostly field-programmable gate arrays (FPGA) and static RAM's – and, as a consequence, runs at a relatively lower speed. In this section we describe the switch of Telegraphos I as example of a possible implementation of the congestion-tolerant network architecture that was discussed above. Several choices of parameters for the Telegraphos I design are due to the technology used – the reader should not interpret them as being inherent characteristics or limitations of the Telegraphos architecture. Section 5 describes Telegraphos I more fully, and it also discusses our plans for the next Telegraphos prototypes.

The Telegraphos I network operates synchronously with one clock of approximate frequency 20 MHz (future prototypes will remove the restriction of synchronous operation and will raise the clock frequency). Each (unidirectional, point-to-point) network link consists of 9 signal wires (and several ground wires), which carry 1 bit of control and 8 bits of data per clock cycle. All network payload travels inside packets (in ATM these are called cells). Packets are of fixed size: $9 \times 8 = 72$ bits (future Telegraphos prototypes will use a longer packet size). In addition to using separate wires (links) for the traffic in the two directions ("forward" and "backward"), Telegraphos I also uses separate wires for packets and for tickets. This waste of resources (since ticket throughput is much lower than packet throughput) was done only in Telegraphos I, in order to simplify the hardware design. Like packet links, ticket links also carry 1 bit of control and 8 bits of data per clock cycle. On a ticket link, the control bit distinguishes between a valid ticket and an idle link. The data part of a valid ticket (8 bits) identifies the VC that the ticket refers to. On a packet link, the control bit identifies the *header* of a packet, and distinguishes it from the body of the packet or an idle link. The data part of a header (8 bits) is the VC number of the packet. The body of the packet (72–8 = 64 bits) is always transmitted on a packet link during the eight (8) clock cycles immediately following the transmission of the header; after these 8 cycles, the link is idle until the next packet header appears. The Telegraphos I switch has 4 incoming and 4 outgoing packet links. Each packet link is paired with an opposite-going ticket link. Since ticket links are separate from packet links, the outgoing links are free to connect anywhere we want – not necessarily to the same places where incoming links come from; this freedom will be lost in the future prototypes, where packets and tickets will be multiplexed on the same links.

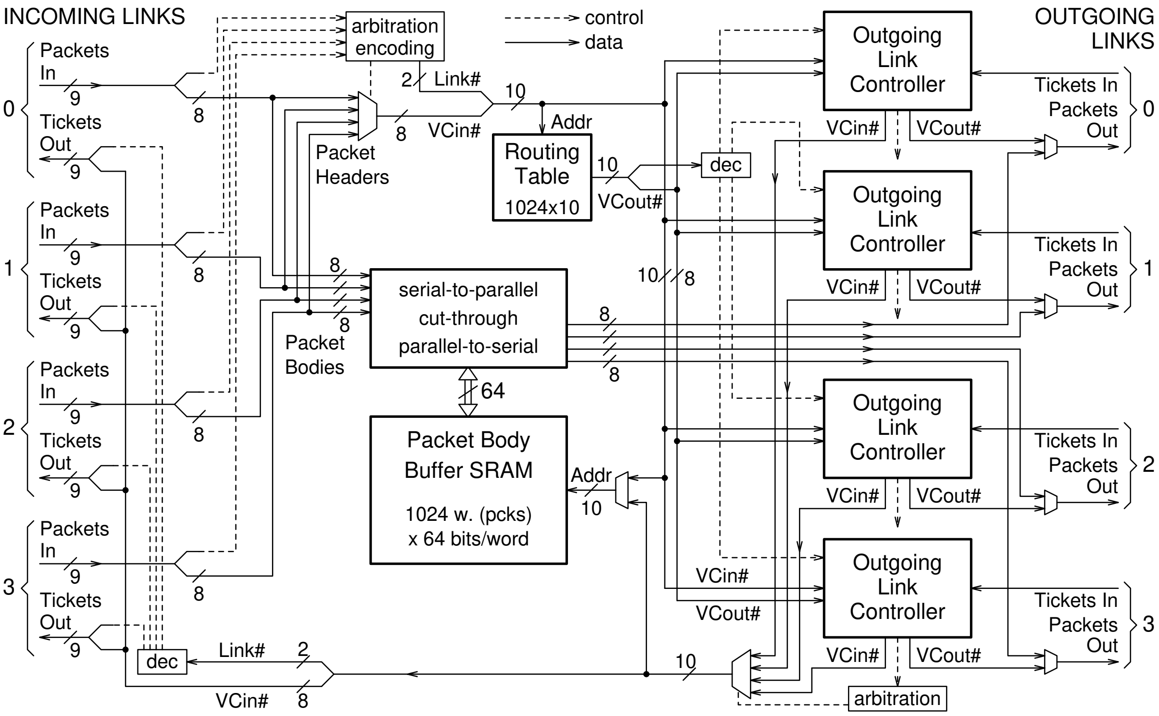

Figure 2.4: Block diagram of the Telegraphos I switch

Figure 2.4 is an abstract block diagram of a Telegraphos I switch. No timing information is contained in this figure, and pipeline registers are not shown. We will describe now the flow of packets through this switch. The control bits of the 4 incoming links are continuously monitored in order to know when new packets start arriving. When this happens, two actions are taken in parallel: *(i)* the packet body is collected in registers that convert it into parallel form (64 bits), and *(ii)* the packet header (incoming VC number) competes against packet headers arriving through other links for access to the (single-ported) routing table. At most 4 packet headers may arrive during any 9 consecutive clock cycles; since the routing table can service 4 requests in 4 clock cycles, we have a known, short upper limit for the waiting time of each header. Routing in Telegraphos I is done as in ATM: the incoming link ID and VC number (VCin# in figure 2.4) of the packet determine, through a table, its outgoing link ID and new VC number (VCout# in fig. 2.4).

Since there are 4 links per switch and at most 256 VC's per link, the Telegraphos I routing table is implemented as a 1024×10 memory (SRAM); it is addressed by the incoming link-VC pair, and it contains as data the corresponding outgoing link-VC pairs. We are examining whether this is the most appropriate kind of routing for parallel and distributed computer systems, and whether we will use it or not in future Telegraphos prototypes.

The packet bodies are buffered in a single SRAM *shared by all ports,* since we found out that this simplifies the switch organization and provides quite high throughput [KaSC91, page 1269]; for future, higher performance switches, we are also investigating cross-point or output queueing. This "parallel" buffer accepts or delivers an entire packet body (64 bits) per clock cycle, so it has enough throughput for the up to 4 incoming packets to be written into it and the up to 4 outgoing packets to be read from it during any 9 consecutive clock cycles. Data I/O into this buffer is performed by the serial/parallel/cut-through block shown in the middle of figure 2.4, whose organization is based on [KaSC91, figure 11]. In a lightly loaded network, the cut-through latency – from the moment the packet header enters the switch to the moment the header leaves – is about 3 to 4 clock cycles. The ticket-packet round-trip delay is 8 to 9 clock cycles (link latency is 1 cycle). So, according to the equation of section 2.2, a buffer space of one packet per VC is enough to achieve full link utilization even with only one busy VC. Accordingly, the size of the packet body buffer is 1024 packets, and the address where each packet is stored is formed by concatenating its incoming link ID and VC number.

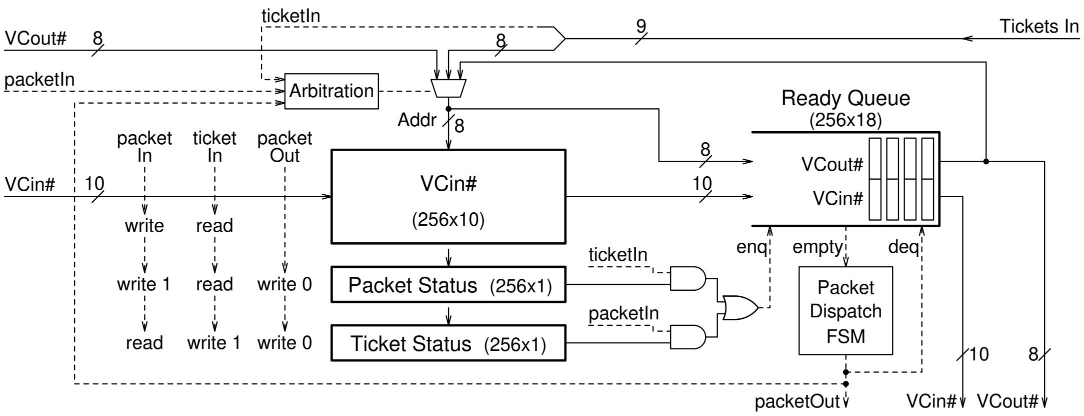

Figure 2.5: Outgoing link controller (one of four in figure 2.4)

At packet arrival time, besides storing the incoming packet in the packet body buffer and translating its input VC number into the output VC number, one more action is performed: the appropriate *outgoing link controller* is notified about the arrival of this packet. There is one such controller for each outgoing link. Its function is to keep track of the packets waiting to go out of this link and the tickets that this switch has for buffers in the next downstream switch, to schedule the departure of packets over this link, and to dispatch the next ready packet when the link is idle. The notification of the outgoing link controller about the arrival of a new packet is done by providing to it a control signal as well as both the incoming and outgoing VC numbers of this packet (VCin# is accompanied by the incoming link ID). The outgoing link controller also receives all tickets arriving from the downstream neighbor. When this controller decides – in the way that we will see below – that it is time for a particular packet to be transmitted over its link, it arbitrates for access to the packet body buffer, and then it sends *(i)* VCout# to the outgoing link, as the new header of the packet, *(ii)* VCin# to the packet body buffer SRAM, as the address for reading the packet body, and *(iii)* VCin# to the appropriate ticket link out to the upstream neighbor, to let it know that now there is buffer space for one more packet of this VC.

The internal organization of an outgoing link controller is shown in figure 2.5. For each of the up to 256 VC's that go out of this link, the presence or not of a packet waiting to go out is recorded in the "packet status" SRAM, and the presence or not of a ticket allowing a packet to go out is recorded in the "ticket status" SRAM. These memories are indexed by the *outgoing* VC number, VCout#, because when tickets come in, they contain that number (the downstream neighbor knows only that). On packet arrival (control signal packetIn), the packet status of that VC is marked as 1 (and its VCin# is recorded for future use), and, simultaneously, its ticket status is read; if the latter is 1, then a ticket for this VC already exists, so the packet is ready and can be transmitted. On ticket arrival (ticketIn), the ticket status of that VC is marked as 1, and, simultaneously, its packet status is read; if the latter is 1, then a packet for this VC already exists, so that packet now becomes ready and can be transmitted. When a packet becomes ready for transmission, its VC numbers, VCin# and VCout#, are enqueued into the "ready queue". This is implemented as a 256×18 SRAM, with two counters as enqueue and dequeue pointers; overflow is not possible, since at most one entry per VC can ever be present on that queue. Whenever the ready queue is non-empty and transmission of the previous packet approaches completion or is already complete, the packet dispatch finite state machine (FSM) dequeues the VC numbers of the next ready packet, and initiates its transmission as described above. It also marks its packet and ticket statuses as 0, indicating that now neither of them exists.

The fact that at most one packet from each VC can be present in the switch at each time, and, consequently, each VC can appear at most once in the ready queue, implies that the VC service policy is *round-robin-like.* This service policy, studied in [Kate87, section IV.C], gives important fairness properties to the switch. In particular, in the presence of congestion, each VC gets an *equal share* of the available throughput, except for VC's that cannot or do not want to use as much throughput as their "fair share"; the throughput that remains unused by the latter VC's is equally distributed to the remaining VC's.

3. Remote-Write Based Message Communication

This section and the next discuss the architecture of the interfaces between the network and the processors. In this section, we deal with the Telegraphos support for message-oriented communication, while section 4 presents the support for virtual shared memory. As we will see, the same basic primitive – the *remote write* operation – is the core of the support in both cases.

3.1 Traditional Network Interfaces and their Problems

Figure 3.1*(a)* is an abstract block diagram of a network interface. We assume an interface between a LAN or a WAN and a computer system – not a processor-network interface in a multiprocessor; however, many of the latter interfaces are strongly influenced by the former, so most of our discussion will concern them too. At least a finite state machine (FSM) is needed, in order to control the transfer of information between the network and the computer. Most of the communication protocols, however, are quite complicated, and thus one or a few FSM's are not sufficient to control their operation. Hence, a processor is used to implement all or part of the protocol. This may be the main processor of the computer, or it may be a separate network interface (NI) processor. The disadvantage of the former solution is that protocol processing consumes an increasing portion of the computer time as the network speed increases. The disadvantage of the latter solution is cost: the NI processor needs a memory system for its operation, and once this and the interface between the two processors are in place we pretty much end up with a two-way multiprocessor where one of the processors is dedicated to protocol processing; for the same cost, we may have preferred a general-purpose two-way multiprocessor.

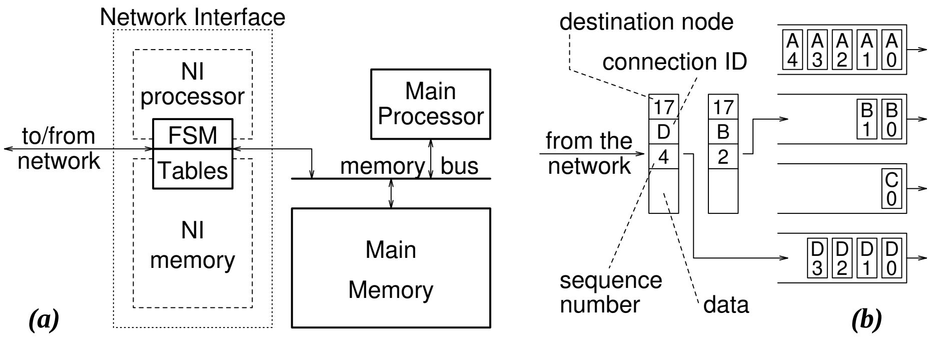

Figure 3.1: Traditional network interface (a) and receiver data structures (b)

The network interface must include some tables containing status and control information. Besides that, buffer space is also needed for the body portion of pending packets and messages; this buffer space is usually much larger than the tables. Again, two options exist: either the main memory of the computer may be used for this purpose, or a (relatively large) memory may be included in the network interface. Both solutions have their disadvantages. Having a separate NI memory increases the cost of the network interface, and the partitioning of the total system memory into NI memory and main memory leads to inefficient use of the total memory space (one may overflow while the other has free space available). On the other hand, using the main memory for packet buffering has another disadvantage in some traditional network interfaces:

consuming memory bus cycles for protocol processing. If the network interface needs to read or write (or read and write, i.e. copy) the packet body more than once between network transmission time and application program send/receive time, then this solution consumes a larger portion of the (scarce) main memory bus cycles than the separate NI memory solution.

Traditional messages passing through a network interface have to go through an entire "stack" of protocols, many of which are implemented in software and are responsible for delays and for degradation of throughput to levels lower than what the hardware can provide. These protocols usually serve the following purposes: *(i)* flow-control, acknowledgements, timeouts, and retransmissions, which, as explained in section 2.2, are handled in hardware or are not needed in Telegraphos; and *(ii)* message break-down into packets, packet re-assembly into messages, message distribution to proper destination application process, and protection of one process' messages from the other. We will examine these latter sources of overhead in more detail now.

Variable-Length Message Reassembly. Handling variable-length packets in the network switches leads to many complications in the hardware – especially in buffer management. Also, given that some messages are very long, as discussed in section 2.3, the network "response time" would suffer seriously if such long packets were allowed and preemptive scheduling were not used; but, preemptive scheduling is equivalent to breaking down messages into packets, and its implementation in hardware is not simple. For these reasons, many high-speed networks (most notably ATM) use fixed-size packets ("cells"); Telegraphos does the same. Once the network packets are not the same as the application messages, someone must break down messages into packets (this is the easy part), and someone must reassemble packets into messages (this is the hard part). Reassembly of packets into messages is hard when packets may be delivered out-of-order (not the case in Telegraphos), and it is also hard because packets of one message are delivered from the network to the NI intermixed with packets of other messages that originated from other sources at about the same time (other processors, or other threads/processes). To handle this second problem, the receiving network interface must buffer packets until entire messages are assembled. As illustrated in figure 3.1*(b)*, this is performed by buffering packets in separate data structures according to the connection ID of each. Obviously, buffer management is quite complicated to be done by a FSM, and thus message reassembly places an overhead on the main processor or necessitates a NI processor. Among other things, buffer management must take care of messages of widely varying sizes, message sizes that may be unknown when the message starts arriving, and buffer space overflow. The policies to apply are not simple or straightforward; for example, what do you do in the latter case, when there is no more space for new message reception (e.g. because receiving applications are slow): do you stop receiving packets, using flow-control, with the danger of deadlocking the network if other processors do the same at the same time?

Arrival Order versus Processing Order (Fixed-Size Messages). Some applications – especially parallel programs in some message-based multiprocessors – exchange short messages or fixed-size messages, exclusively or in majority. These messages fit in one network packet each, so there is no overhead for reassembling messages from packets. Still, another related overhead remains. If packets are no longer placed into message queues as in figure 3.1*(b)*, it means that packets (messages in this case) are no longer sorted by category (destination process, thread, or connection ID). Instead, the network interfaces in these multiprocessors usually place all arriving packets into a single FIFO queue, and deliver them to the processor in strict arrival order. However, arrival order may often not be the desired processing order, so there is now a software overhead to properly categorize and process messages.

Protection – System Call Overhead. A computer system needs to protect one process from another – at least to protect one program from the bugs of the other, if not to protect its users from malicious intruders. Since the network interface is an I/O device, its use must be controlled by the OS. To send a message, parameters must be written into control registers of the NI; this must be done through a system call. At the receiving end, the user-level program that happens to be running when a packet arrives cannot be trusted to receive all packets, including those destined for other processes: if it is buggy, other programs would malfunction. Hence, packet reception must be done in system-mode, through interrupts. Interrupts and system calls are expensive, and this is another cause for the high overhead of traditional message passing.

Message Copying. Copying of the message bodies from one place in one memory to another place in the same or another memory leads to the consumption of memory bus cycles or the need for separate NI memory, as discussed earlier in this section, and also leads to delays in message delivery. Message copying is another manifestation of the two previous problems. First, during packet to message reassembly, it is hard for the buffer(s) chosen by the network interface to happen to be the same as the memory location(s) that the application program expects to see the arriving message in; hence the message must be copied from one place to another. Second, traditional message passing protocols have several layers and several protection boundary crossings: from application program to message server, to kernel, and back. At each boundary crossing the message is often copied from one address space to another. Although everyone knew that copying was not the best thing to do, people were willing to live with it, at least in the past, for two reasons. First, network transfers were very slow a few years ago; thus, memory to memory copy was a small percentage of the message transfer time. And second, older operating system kernels were monolithic, which implied only one protection boundary crossing for message passing at each end (user to kernel); however, recent microkernel-based operating systems that use user-level servers introduce at least one more crossing at each end – we may now have user-to-server and server-to-kernel boundaries.

3.2 The Remote-Write Communication Primitive

Message passing in a *single address space multiprocessor* is much simpler than between computers communicating over a traditional LAN or WAN or between the processors of a "message passing" (multiple private address spaces) multiprocessor. The usual form of single address space multiprocessor – what is usually called a (hardware) shared-memory machine – is expensive, because it strives to provide high-speed *read* accesses from remote memory locations; this issue is discussed again in section 4. However, in order to support message passing efficiently and at low cost, we only need a subset of the functionality of such a machine: the *remote write* operation. Network interfaces that support this operation are simple, inexpensive, and they solve the problems of traditional network interfaces that were listed in section 3.1. Hence, Telegraphos uses such a network interface; as mentioned at the beginning of the paper, the very name of the project means "remote write" in Greek. We will first describe the remote write operation, and then explain how it solves the problems of section 3.1.

Figure 3.2 illustrates the remote write operation. In a single address space system, each physical address refers to a particular word in the local memory of *one* particular processor (computer) on the network; in other words, the physical address space is shared among the processors. Thus, any virtual address generated by any processor on the network may refer, after address translation, either to a word in the local processor's memory or to a word in any other (remote) processor's memory. This is accomplished by treating the MS bits of every physical page number as a processor identifier, and the LS bits as a page number in that processor's memory. In this section, we only deal with *store* instructions into addresses that translate into remote memory locations; see section 4 for load instructions. From the implementation point of view, when the virtual address of a store instruction translates into a physical address in the Telegraphos device I/O space, the Telegraphos network interface places the address and the data into a packet, and sends this packet through the network to the destination identified by the MS bits of the physical address. At the destination node, the Telegraphos interface performs the (local) memory write.

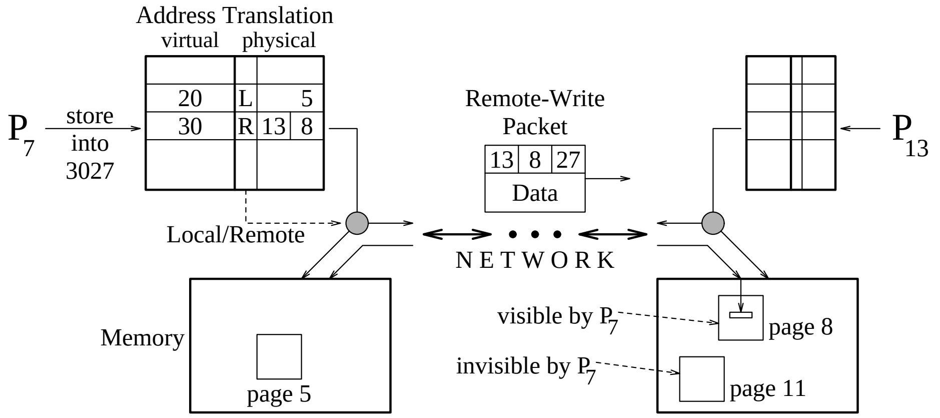

Figure 3.2: Processor-network interface: the remote-write operation

A remote write packet traveling through the network can be thought of as an extension of a usual network packet. In order for message reassembly to be possible, the usual network packets include a message (connection) identifier, and a packet sequence number. The remote write packet simply extends these identifiers by a few bits and makes them into a full memory address in the destination processor's memory. The other half of the story is that a usual network packet is transmitted by an I/O operation issued in system-mode, while a remote write packet is transmitted by a store instruction executed in any processor mode, including user-mode.

In a single address space system, a message is sent by a sequence of remote write operations: the sender writes the words of the message into specified addresses in the local memory of the receiver. Signaling the completion of a message send operation is a case of parallel program synchronization. In Telegraphos, where packets from one source to one destination are delivered in-order, the last packet of a message may be such that e.g. it sets a flag that was previously zero, in a previously agreed memory location. Let us see now how sending messages by remote writes solves the problems of traditional network interfaces listed in section 3.1.

Low Cost Network Interface. The transmitter part of the network interface has to place address-data pairs into packets and send them through the network. The receiver part of the NI has to request use of the local memory bus in order to perform write operations into the main memory. Both parts are so simple that small FSM's are enough to implement them: no NI processor is needed, and no cycles of the main processor are consumed either. Second, communication is performed using the normal "main" memories of the processors: no NI memory is needed either. Thus, the network interface cost is minimized.

Long Messages are Assembled In-Place. Since each packet contains its destination memory address, received packets are automatically segregated into messages: each goes into its own (preallocated) message buffer. It does not matter how many remote write packets it takes to send a message, neither does it matter whether these arrive in-order or out-of-order, interleaved with other unrelated packets or not. Since each of them carries its own destination address, each goes and places its own "pebble" at the proper place of the final "mosaic" (the message). Independent of when exactly each pebble is placed, they are all in place when the message is complete. Buffer allocation and management becomes the responsibility of the communicating applications: they must first agree on the memory space to be used for communication, and then they can send messages that are delivered directly into the desired memory area. There are well-known software methods for (pre-) allocating memory in programs, for managing communication buffers, and for using enough buffer space to support the desired degree of pipelining in the application's operation – the network interface hardware does not need to do that. Since buffer space for each message has been preallocated by the application, *no buffer overflows* may ever happen.

Processing Order is Independent of Arrival Order. The remote-write mechanism delivers each message directly into its appropriate "mailbox", without processor intervention. The message will stay there until the processor decides to process it. Thus, the processor is no longer forced to handle packets/messages in strict arrival order – not even in order to place them into the appropriate queue for further processing; it can now process them in the order that it desires.

Memory Mapping implements Message Protection. A user process can only access those physical memory pages that the operating system has mapped into its virtual address space; this is true both for local and for remote pages. Since message sending in Telegraphos is accomplished by executing (remote) memory store instructions, a process S can only send messages to those processes R, and into those mailboxes of theirs, for which the OS has given S an entry in its translation table. Similarly, a process can only receive (read) the messages that were sent into pages that it has access to. As a consequence, message exchange can be done strictly *at user-level,* without need for any time-consuming system call. In this way, an *already existing* mechanism – address translation – that is used to implement memory protection, is also exploited in order to implement communication protection.

No Message Copying: since messages arrive "in-place", there is no reason for copying them from place to place. The parallel application program is free to choose the best place in the receiver's address space where the message should arrive, and then the sender can simply send it right there. The choice may be made by user code or by library routines, or by the compiler, or by the run-time environment – all that matters is that the software is in control.

3.3 Filling Up Multiword Packets: Hardware Packetizer

The discussion above omitted one issue. Traditional network packets are transmitted by an I/O operation, which can specify their contents regardless of how large these contents are. On the other hand, remote write packets are generated by store instructions, which (in modern RISC processors) can only specify one word of data. If network packets are larger than one address-data word pair, then how are they filled – or are they always transmitted half-full?

A simple hardware circuit, which we call a *packetizer,* can undertake to fill packets that are larger than one address-data pair. The key idea is that every store instruction into a remote address does not need to trigger an *immediate* transmission of a remote write packet. Instead, remote writes can be buffered, compared against each other to see if their destination addresses are neighboring, packed together, and then transmitted when one of the following happens: *(i)* enough neighboring remote writes have been collected to fill up one packet, or *(ii)* too many non-neighboring remote writes have been buffered and we thus run out of buffers for partially-filled packets, or *(iii)* the processor requests to flush all buffers and immediately transmit all pending remote writes, because of an imminent synchronization point in the parallel program.

For message support purposes, a packetizer with just one partially-filled packet buffer is enough. Assume that the packet contents are the following: a destination address A_0 (its MS bits specify the destination node ID), the remote write opcode, a count n of words to be written ($1 \le n \le N$), and N data words, $w_0, w_1, \ldots w_{N-1}$. This packet means: $M[A_0] \leftarrow w_0$, $M[A_0+1] \leftarrow w_1, \ldots M[A_0+n-1] \leftarrow w_{n-1}$ (assuming word addresses). The packetizer includes buffer space for the contents of one packet: A_0, n, and $w_0, w_1, \ldots w_{N-1}$. When the processor issues a remote write operation for word w to be written into address A, the FSM of the packetizer performs the following: (1) if $n \ne 0$ (the packetizer is non-empty) and $A \ne A_0 + n$ (the new remote write is not to the "next neighboring" location), then transmit the current contents of the packetizer in a partially filled packet, and do $n \leftarrow 0$; next (2) if $n = 0$ (the packetizer is empty), do $A_0 \leftarrow A$; next (3) do $w_n \leftarrow w$, and $n \leftarrow n+1$; next (4) if $n = N$ (the packetizer became full), then transmit the current contents of the packetizer in a full packet, and do $n \leftarrow 0$.

As long as one thread of one process executes store instructions into sequential remote addresses, this packetizer will break these up into groups of N words, and will transmit each group in one network packet, as soon as the group is complete. The last group of $< N$ words will wait in the packetizer. When this or another thread of this or another process executes store instructions into a non-sequential remote address, this last group of remote writes will be transmitted, and a new group will start being formed. Thus, in the usual (under message passing) case of stores into sequential remote addresses by an uninterrupted process, full packets are transmitted, except for the last packet in each message. Correctness of operation of the packetizer is independent of interrupts and context (process) or thread switches, since it operates with physical addresses, and since such switches will normally cause a break in the sequentiality of the remote addresses, thus triggering transmission of partially filled packets. Finally, any user process can write into a special control address of the network interface which is reserved for signaling a request to flush the packetizer; if $n \ne 0$, this will trigger immediate transmission of the partially filled packet currently residing in the packetizer. No protection is needed for this special control address, since writing into it is harmless for correct operation – the worst that can happen by needlessly writing into it is performance degradation.

4. Shared Memory Support and Eager Updating

As we saw in section 3, a shared address space and the remote write operation are vital for high performance message communication at low hardware cost. In this section, we will see how these same basic mechanisms, augmented with additional remote memory operations and simple hardware monitors, can also support the shared memory model of parallel computation. Even though replication is only at the page level of granularity, this style of parallel architecture is quite promising for the future, owing to its low cost, and because it places the software in control of the important but ambivalent decisions: accessing remotely versus replicating locally, and updating versus invalidating.

4.1 Options in Managing NUMA Shared Memory

You can get all the memory to be equally slow, if that is what you want,
but you cannot get it all to be equally fast. – Anonymous

Shared memory – more formally defined as *single address space* – is widely accepted as the most convenient machine model for the parallel programmer, since all threads of computation have random access to all data structures at any time. There is one more requirement for shared memory to really be convenient: all threads should have *equally fast* random access to all data structures – otherwise, the programmer has to worry about data and thread placement, and convenience goes away. Unfortunately, such uniform memory access (UMA) systems suffer from the well known principle quoted above: if all memory has equal access time, then this common access time is that of the slowest part of memory. Since there are only advantages in having some part of the memory be faster than other parts – rather than having all parts be equally slow to the slowest part – large shared memory multiprocessors implement *non-uniform memory access (NUMA).* In large multiprocessors, the difference in access time between the nearest memory (local processor's cache) and the furthest memory (distant processors' main memory) is quite large or even huge. Thus, there are considerable factors of performance to be gained by being conscious of the memory hierarchy, and by using and controlling it in the proper way. This can be done by hardware at run-time, by software at run-time (operating system, run-time environment), by software at compile-time (optimizer), or by the programmer at algorithm development time.

In several cases, in a NUMA multiprocessor, it is advantageous to make and use local copies of information – that is, to *replicate information* – rather than accessing it remotely, i.e. from the slower parts of memory. This is the well known and proven mechanism of *caching* in *memory hierarchies.* This is *not* to say that *all* information should be cached locally prior to being used – there are cases when it is better to access some information remotely rather than replicating it locally. Replication of information does incur overhead: directories are needed to keep track of the copies, and the original and the copies have to be maintained consistent with each other (the "memory/cache coherence problem").

Granularity of replication is a fundamental issue in designing a caching scheme: should copying be performed in units of *cache lines or memory pages?* Several factors have to be traded off against each other. The larger the unit of replication, the longer the fetch latency and cost, and the more the problem of false sharing manifests itself. On the other hand, the smaller the unit of replication, the larger the directory costs, and the less the prefetching effect becomes. At the two ends of the spectrum of choices we have the two basic shared memory systems:

Coherent cache systems replicate information in units of cache lines, i.e. relatively short blocks. Since the latency of copying cache lines is short, and the frequency of accessing new cache lines is high, replication is handled by *hardware,* because it has to be fast. Cache coherence protocols implemented over *bus* networks became quite popular for small-scale multiprocessors – see e.g. [SwSm86]. However, for large multiprocessors, *directory based* protocols are needed – see e.g. [Leno90], [SCI92]. Directory based cache coherence protocols have significant design complexity and implementation cost. They are in general considered as a good but expensive approximation to the "shared memory illusion". Let us add here that, since they are implemented in hardware, they offer little or no room for the software to guide their operation – something that may prove important in the future.

Virtual shared memory systems replicate information in units of main memory pages, i.e. relatively large blocks – see e.g. [LiHu89]. As with traditional memory-disk paging, given the relatively high cost of copying an entire page, replication is managed by *software,* since this can now be tolerated and it is simpler. Virtual shared memory, in its simplest form, can be implemented purely in software, on top of a network of uniprocessors; thus, its hardware cost can be null. The main idea is that all copied pages are marked read-only; when a processor attempts to write into such a replicated page, the page-fault that results invokes the operating system, which invalidates all other copies of the page before allowing the write to proceed by marking the (now single copy of the) page read-write. However, such software-only implementations of virtual shared memory have a poor performance, mostly due to the large degree of false sharing and to the frequent copying and invalidations of such large blocks. As we will discuss below, however, there are a number of relatively inexpensive hardware facilities that, when present, can significantly improve the performance of virtual shared memory, and can thus make it into a serious competitor of directory based cache coherence. Additionally, virtual shared memory can exploit an extensive set of software guidance and optimizations, which can potentially make it even faster than "blind" hardware cache coherence, in the future. This is the direction taken by the Telegraphos project.

Segregation of data into blocks of replication, so as to reduce false sharing is of outmost importance, as our studies [MaCh94a] confirmed. The compiler (and perhaps the run-time environment), possibly assisted by programmer hints, should avoid to place data structures with different sharing patterns in the same replication block – cache line or memory page. This is a very important direction for future research in parallel program optimization, especially when replication blocks are as large as pages.

Replicating locally versus accessing remotely is a fundamental choice to be made for each access to each data structure in a NUMA hierarchy. As mentioned earlier, the fact that remote information can be replicated (cached) locally does *not* imply that *all* information should be cached locally prior to being used. The principal factors involved in making a choice are: *(i)* is the access a read or a write? – writes are tolerant to latency; *(ii)* can the read access be foreseen far enough in advance so as to prefetch? *(iii)* what is the cost of copying an entire block relative to making a single word remote access? *(iv)* will the cost of copying an entire block be amortized by the subsequent local accesses to its contents? and *(v)* what will be the cost of maintaining the copy consistent with its original? In some cases, choosing between local replication and remote access is fairly easy and simple – e.g. read-only information should be replicated. In other cases, the choice is not as easy in general, but it may boil down to some simple rules in practice – e.g. locks should usually *not* be replicated. In many other cases, the tradeoffs among the above factors are complicated, with little predictability, and the choice cannot be based on such simple criteria as above. As in many similar cases in computer architecture, what this means is: *avoid making a decision in hardware – let the software decide.* In coherent cache systems, this can be achieved by *hints* supplied by the software to the cache hardware. In virtual shared memory systems, the code that manages replication – the run-time environment and the OS – can also make the above decision. When letting the software make decisions, it is very desirable for the hardware to provide *measurements,* to allow the software to make an informed decision.

Updating versus invalidating the other copies *upon writing* into one of them is the second fundamental choice to be made for each write access into a replicated block in a NUMA hierarchy; one of the two actions is necessary in order to preserve coherence of the copies. It is better to invalidate a copy when that copy will no longer be used, or when it will be used again after many more modifications to its contents. On the contrary, it is better to update a copy when it will be used again after a modest number of modifications. The break-even point depends on the size of the block under question, and on the total amount of network traffic generated by shipping all the updates versus reshipping the entire block. Due to the complexity of the tradeoff, as before: avoid making a decision in hardware – let the software decide. Besides the above consideration to minimize the total network traffic, *eager updating* – [WiMa89], [Mapl92] – has another advantage over invalidating and recopying at a later time. If the copy of the memory block is needed again, and if this need is not foreseen far enough in advance so as to prefetch the block, then the invalidation method will result in stalling the application program when the latter tries to read this memory block after the local copy has been invalidated. On the contrary, eager-update communication is performed in parallel with ongoing computation. Thus, updating may be preferable over invalidating *even if* updating causes more total network traffic than recopying the needed memory blocks. In other words, eager updating *trades network throughput for less strict latency requirements.* As the cycle times of IC pipelines become shorter and networks become "more pipelined" (higher throughput but more latency), eager updating will become more important, for precisely the reason stated by the following quotation....

There is an old network saying: Bandwidth problems can be cured with money.
Latency problems are harder because the speed of light is fixed – you can't bribe God.
David Clark, MIT

4.2 Shared Memory Support in Telegraphos

Like virtual shared memory systems, and unlike coherent cache systems, Telegraphos supports replication of information under software control, at the page level of granularity. We made this choice because of its significantly lower design and implementation cost, and because it provides a more appropriate and challenging environment for developing software methods for optimized data and thread placement and for managing NUMA hierarchies accordingly. However, unlike software-only virtual shared memory implemented on a network of uniprocessors, Telegraphos provides a number of hardware facilities that are relatively inexpensive to implement, but can significantly improve the performance of the baseline virtual shared memory. These hardware support mechanisms for shared memory are the following.

Single address space, rather than the multiple private address spaces provided by distributed systems consisting of networked uniprocessors, is the basic "umbrella" of these mechanisms. In this way, the software has a choice between accessing remotely and replicating locally. In multiple private address space systems, the only way to access remote memory locations is via a page fault that leads to page replication, or via something like an RPC that involves interrupts and OS intervention. In Telegraphos, remote access without replication is possible; this may be advantageous in a number of cases, as detailed in the next few paragraphs.

Remote write operations (seen before in section 3.2) form the basis of the Telegraphos architecture; they also form the unifying link between message passing and shared memory. Writing into a remote page can be, in some circumstances, as fast as writing into a local page; thus, in the case of writes, there is no a priori reason to replicate the page locally. This situation is analogous to write-through in cache hierarchies, and to the no-allocate on write-miss policy. In Telegraphos, remote writes are non-blocking: the processor does *not* wait for an acknowledgement that the operation arrived at its destination and was executed (in fact, there are no such acknowledgements in the Telegraphos hardware). Detection of the completion of all previous remote writes before a point of synchronization in a parallel program is provided by separate, collective methods. Thus, the latency of a remote write, as seen by the processor, is the same as the latency of a write operation into a local I/O device; this is slower than a write into the local cache memory, but the write buffer that most modern processors have masks this difference. The throughput of consecutive remote writes is obviously the throughput provided by the interconnection network, which is usually lower than the throughput of the local memory. Thus, if the processor performs many consecutive remote writes, the write buffer will soon be filled and further remote writes will be delayed by stalling the processor. In summary, remote writes are as fast as local writes, provided that no more of them than the depth of the write buffer are issued in any given short period of time (equal to the write buffer size divided by the network throughput).

Remote read operations are provided in Telegraphos, but they are slow when the processor does not support continued execution while load instruction completion is pending; a round-trip delay through the network is needed before the data can be returned to the processor. Still, given the high cost of replicating locally an entire remote page, remote reads are preferable when there are no reasons to believe that the cost of replicating the page will be amortized by repeated read's from it before it is invalidated. When the software can foresee a certain read access before it is made, and when it can foresee that this will usually be a remote access, then it should rather use a remote copy operation, as described next.

Remote copy, or *prefetch,* or *non-blocking read* operations are the counterpart of remote writes in allowing the software to overlap communication and computation. The remote copy operation takes two addresses, A_{from} (remote) and A_{to} (local), as arguments. Its execution causes transmission of a packet to the node containing A_{from}; there, the data read from A_{from} are placed in a remote write packet with destination A_{to}. Normally, A_{from} is expected to be remote, and A_{to} is expected to be local; however, local A_{from} is also allowed, because it may not be known at compile-time whether a local copy of this remote page exists at run-time. The remote copy returns control to the processor without waiting for completion of the operation ("non-blocking" read). To detect completion, the processor must have written an illegal value into A_{to} before issuing the remote copy; when a legal value is found in A_{to}, it means that the remote copy was completed. Collective detection of completion is possible, when the network uses static routing and all remote copies were from the same remote node. Issuing a remote copy operation is more complicated that merely executing a load/store instruction (as for remote read/write). In Telegraphos I, three instructions are needed: one to place the network interface into remote-copy special mode, one dummy read from A_{from}, and one dummy write into A_{to}; execution of the two latter instructions performs address-translation and access rights checking. This three-instruction sequence must not be interrupted, so we perform it in DEC Alpha's "PAL code".

Remote atomic operations are also supported by Telegraphos, as an efficient synchronization primitive. They take one address and one or two data words as arguments. If the address is remote, the operation code and the arguments are placed in a network packet and are sent to the appropriate node. At the destination node, the specified operation is performed atomically on the local memory. If the operation was remote, the return value is sent back in a network packet. The issuing processor is blocked until the return value is given to it, as for remote read's. Telegraphos I implements the fetch-and-store, fetch-and-add, and compare-and-swap atomic operations. The target page of an atomic operation must *not* be replicated (no other copies of it must exist). Issuing an atomic operation is done by executing two or three instructions, similar to remote copy. Implementing atomic operations remotely, rather than replicating or migrating locally the block that contains the synchronization variable, is a big saving of time, especially in page-oriented shared memory – but oftentimes in coherent-cache machines as well. Operations on (and, consequently, "ownership" of) synchronization variables alternates often among several processors. In a virtual shared memory environment, this implies that migration of an entire page in order to perform the operation locally is grossly inefficient. Even in a coherent-cache machine, if processor A wants to acquire a lock that normally resides in B's memory but was last owned by processor C, then at least 3 or 4 network transitions are needed for A to ask B, B to notify A and/or C, and C to migrate the block to A; more costly and complex situations arise when lock contention by multiple processors occurs. In contrast, if the lock never moves, A can perform a remote atomic operation on it (at B) in only 2 network transitions, as in Telegraphos.

Monitors and alarms for the number of remote page accesses are implemented in the Telegraphos hardware. The runtime environment and the operating system would like to be able to make informed decisions about when to replicate a remote page locally. This basically depends on the frequency of read accesses from this processor to that page, relative to the frequency of writes (from all sources) into that page. Every Telegraphos network interface maintains (in a RAM) two counters per page – one for read's and one for write's. Each incoming or outgoing remote access operation decrements the corresponding counter (unless the counter was zero).

When a counter goes from 1 to 0, the processor is interrupted. By setting these counters to very high values and then periodically reading them, the software can monitor the number of remote accesses to each page. On the other hand, by setting them to relatively small values, the software can implement alarm-based replication algorithms: when the number of remote accesses to a certain page exceeds a predetermined value, the software is notified in order to make a corresponding replication decision; [MaCh94a] and [MaCh94b] present our studies on this topic. A counter can be disbaled by setting it to zero. The cost of the counter RAM is not very high, because its throughput needs not be high: only one operation per network packet must be performed.

Eager updating of remote copies – [WiMa89], [Mapl92] – is provided in hardware in Telegraphos I: writes into a page are automatically reflected into all copies of the page that may exist on multiple nodes. The Telegraphos network interface includes a directory (kept in an SRAM) listing, for each local page, all remote copies of it. For each write access to one such local page, the interface generates and sends (multicasts) one remote write operation to each of the remote copies of the page. The contents of the directory are under the control of the operating system and the run-time environment; thus, the decision on which of the pages should have their copies updated is made by the software. As discussed in section 4.1, there are several cases when updating the copies of the page being modified is much better than invalidating them. For example, consider a page that is "read mostly" – it is often read and rarely written; it would be grossly inefficient to invalidate and then recopy the *entire* page after every write operation into it. Thus, updating can lead to less network traffic, but this is not the only thing that matters: updating can be preferable even when it leads to more network traffic, because it overlaps computation and communication better, and thus it leads to less processor stalls on read misses. By providing automatic multicasting of the updates in hardware, Telegraphos allows the software to take advantage of eager updating over invalidating, when the software believes that this is the best thing to do. Eager updating extends the remote write – the basic Telegraphos operation – so that this operation writes into all copies of its destination.

This concludes the list of Telegraphos support for shared memory. In brief, the Telegraphos architecture aspires to bridge the gap between message passing and shared memory, by providing a relatively simple hardware mechanism that can be used *both* to support efficient message passing, and to provide a good intermediate point in the spectrum of shared memory architectures, between fast but expensive coherent caches and inexpensive but slow virtual shared memory. This basic primitive in Telegraphos is the *remote write operation,* in a *single address space* environment. For each page, the compiler, the run-time environment, or the OS can choose local replication or remote accessing; in case of replication, a choice of coherence protocol is provided between write-invalidate and write-update. These choices – static or dynamic – are assisted by hardware monitors and alarms, and they may prove, in the future, better than "blind" coherent caches.

4.3 Implementing Telegraphos as a Workstation Farm

One of the most interesting aspects of Telegraphos is that it blurs the boundaries between distributed systems (uniprocessors on a local area network) and multiprocessors. Independent computer systems, like workstations, that were designed as uniprocessors, can be linked via Telegraphos into something more than a "workstation farm" – into a true parallel computer. This was shown in section 3, which explained how Telegraphos supports fast message passing, and in this section, which showed how Telegraphos supports shared memory. Figure 4.1 illustrates how the Telegraphos network interface must be connected to a computer system in order to achieve most or all of the above functionality.

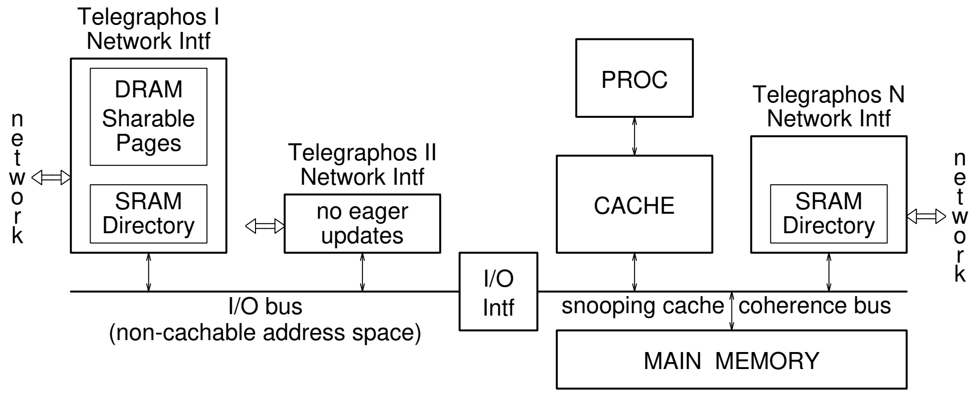

Figure 4.1: Interfacing Telegraphos I, II, and N to existing workstations

Normally, the network interface of Telegraphos must be connected on the memory bus of the processor, which must run a snooping cache coherence protocol. This is illustrated in figure 4.1 as "Telegraphos N", indicating that future Telegraphos versions should connect there. For the moment however, Telegraphos I and II, the two current prototypes, only connect to an I/O bus (DEC Alpha's Turbochannel for Telegraphos I); this was done for purely practical, economic reasons: Alpha workstations with connection slots on a documented cache coherence bus were unavailable or very expensive for our budget. The result is that Telegraphos I implements eager updates but lacks caching on sharable data, while Telegraphos II lacks the former but has the latter, as explained below.

The most difficult part of the processor-Telegraphos interface is the multicasting of the updates to replicated pages, when a local copy of the page exists, and the word that is being updated resides in the processor's cache. In order for the Telegraphos interface to see this update and to be able to multicast the new value to the other nodes holding copies of the page, Telegraphos must reside on a cache coherence bus, and we must arrange for the update to become visible on that bus. Once Telegraphos is on a cache coherence bus, it can pretend that its "cache" has a copy of all lines belonging to shared pages (those pages appearing in the directory of the Telegraphos interface). Consequently, any such line that appears in the processor's cache will be in a "shared" state. It follows that, whenever the processor modifies such a line, it has to broadcast an update or an invalidate command for that line on the memory bus; this will be seen by the Telegraphos interface. If it is an update command, Telegraphos knows the address and the new data, and that is all that is needed for eager updating (multicasting) to work. If on the other hand the bus runs an invalidate-based protocol, Telegraphos will see the address of the invalidated block, and it will ask to read that block immediately thereafter; then, it can multicast its new contents.

When the Telegraphos interface has to plug into an I/O bus, as in Telegraphos I and II, write operations by the processor into pages residing in main memory are *not* visible by Telegraphos, since they do not appear on the I/O bus (either they hit in the cache and thus do not appear on the memory bus, or else the I/O interface filters them out). Consequently, a replicated page cannot reside in the processor's main memory, because, if it did, it would be impossible to eagerly update the remote copies when the local processor modifies its own copy. For this reason, Telegraphos II will offer no eager update (multicast) operations. All other remote operations (remote write, remote read, etc.) can still work fine, because the OS can map remote pages into the I/O physical address space (we assume memory-mapped I/O). The processor's TLB, cache, and I/O interface collaborate to make I/O-space addresses non-cachable. Thus, all write's and read's by the processor to/from this space become visible on the I/O bus and by the Telegraphos interface; the latter, then, sends them through the network.

Telegraphos I follows a different approach in order to allow us to implement eager updates even with an I/O-bus based interface. Telegraphos I includes DRAM on the network interface itself. All sharable pages (pages visible through the network) are mapped into this DRAM. Thus, all operations by the local processor onto these pages are visible by the Telegraphos network interface, so that write's into replicated pages can be multicast through the network. The disadvantage, in this case, is that sharable pages are non-cachable, and thus accesses to them by the local processor are slow; the other disadvantage is that the Telegraphos network interface becomes more expensive, since it has to include DRAM on board. For these reasons, we do not recommend the Telegraphos I organization for future versions; however, it is good enough as a first prototype, since it enables us to test the Telegraphos architecture and to collect measurements on the operation of the system and on the efficiency of eager updates (the Telegraphos I network interface includes address trace-collection hardware).

5. Current and Future R&D Activities

Telegraphos is a large project, involving many people, and including architecture work, hardware prototype building, and systems software implementation. This paper only describes its basic architecture. Work on particular architectural aspects of it is ongoing, and papers describing them will be forthcoming. This section is a short overview of the Telegraphos hardware prototypes, of its relation to standards and other projects in the area, and of our architecture research topics.

5.1 The Telegraphos I and II prototypes

Telegraphos I, our first prototype, is currently in the last phases of its hardware design, and it will be built soon. It will interconnect DEC Alpha [Sites93] workstations, plugging into their Turbochannel I/O bus, as discussed in section 4.3, and as illustrated in figure 5.1. Emphasis is on rapid prototyping, so the technology used is conservative: FPGA's, SRAM's, and TTL buffers; clock frequency will be about 20 MHz, and link throughput will be about 160 Mbits/s in each direction (8-bit wide links, in each direction). A 4-by-4 switch will occupy one board; this board will include 110 Kbits of single-ported SRAM plus the approximate equivalent of 4,500 gates of random logic (counting one flip-flop as one gate) in FPGA's. A network interface will also occupy one board; this will include 16 MBytes of DRAM, where the shared data will be placed (figure 4.1), a few SRAM's and dual (asynchronous) ported FIFO's, and on the order of 5,000 gates of logic in FPGA's. Systems software is under development: the Mach operating system is being adapted to run on Telegraphos.

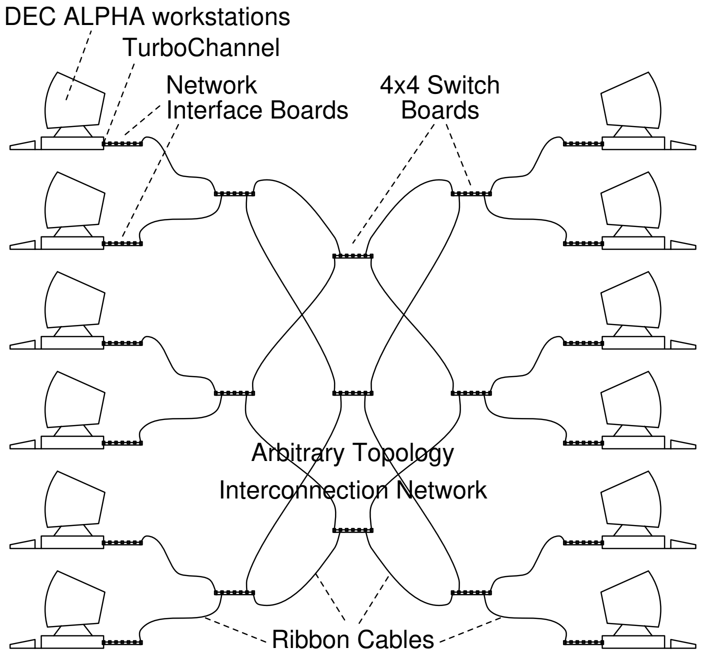

Figure 5.1: Example of a Telegraphos I system

Work on Telegraphos II is starting now. CMOS standard-cell ASIC technology will be used to reduce the size and increase the speed of Telegraphos I. Links should run in the range of 500 Mbits/s, and a 4-by-4 switch will be integrated in a single chip; another chip will implement a processor-network interface (without eager sharing, for the reason explained in section 4.3; external SRAM will be used for the page access counters). The technology envisioned is 1 μm CMOS by ES2. For other future Telegraphos versions, we are considering designing key sub-systems in full-custom CMOS, and using various techniques in order to achieve much higher performance.

5.2 Applications, and Relation to SCI, ATM, and Others

Telegraphos has numerous applications in several different areas. The Telegraphos switches, or other similar switch chips incorporating architectural ideas from Telegraphos, can be used as building blocks for parallel computer networks, for high speed local area networks, or for switching centers for wide area networks. The Telegraphos network interface can be used as a model for low cost and high performance interfaces between high speed networks and computer systems. Thus, Telegraphos is appropriate for a wide range of systems – from parallel computers and multiprocessors, through "workstation farms", to general data networks.

The use of multiprocessor communication components for building high-speed local area networks seems to emerge in several contemporary projects and products. For example, the "ATOMIC" project at USC/ISI uses "Mosaic" switches from Caltech to prototype a gigabit LAN testbed [CFFS93]. This technology is recently being commercialized by Myricom, Inc., under the name "Myrinet". The network interface of Myrinet uses a processor dedicated to network protocol processing; no remote writes and other similar remote operations are supported. Also, Dolphin has recently introduced a commercial product, the "SBus-to-SCI Adapter Card" [Dolp94], which is based on the ANSI/IEEE Standard 1596-1992 "Scalable Coherent Interface" [SCI92], and which interconnects workstations through a 1 Gbit/s ring. This SBus-to-SCI Adapter implements remote writes and reads; for the moment, only the baseline SCI network topology is available: a ring. Network packets containing destination memory addresses and remote write/read commands are also used in the Meiko "CS-2" multiprocessor (without implementing cache coherence among the processors) [HoLa93].

Our current Telegraphos prototypes use proprietary link-level protocols. However, this was only done for simplicity, and because our focus was to investigate some fundamental issues in high-speed switch architectures, which are similar in all such kinds of switches. In the future, the Telegraphos architecture may well be adapted to existing or emerging high-speed network standards, like SCI – mentioned above – or the Asynchronous Transfer Mode (ATM). In SCI, the current focus is on rings, but switches will soon be needed for higher throughput. Regarding ATM, Telegraphos already has a fixed packet (cell) size, like ATM does. In future Telegraphos switches, packets will need to be longer than the current 9 or 16 to 32 bytes of Telegraphos I and II; thus, the 53-byte packet size of ATM will be the natural choice for Telegraphos too, as in our previous work [KaSC91].

5.3 Current and Future Architecture Research Topics

Several of the crucial aspects of the Telegraphos architecture were mentioned in very abstract form in the previous sections, due to lack of space, and because work on them is still ongoing. Here follows a list of these topics.

- *Switch queueing and scheduling* organization for high-speed: The Telegraphos I switch uses a straightforward shared buffer organization for the packet bodies (section 2.5). For switches of much higher performance (more links and faster links), we are working on various appropriate queueing and scheduling architectures.
- *Merging/diverging virtual circuit/path* architectures: As mentioned at the end of section 2.4, in large networks, with many VC's, it would be quite expensive to dedicate one buffer to each VC. We are currently working on architectures that address this problem by grouping together virtual circuits into virtual paths over the common part of their route.
- *Static versus dynamic routing* of packets through the network: Telegraphos currently uses static routing, which has the desirable side-effect of guaranteeing in-order packet delivery from one source to one destination. However, there are cases where static routing results in underutilization of network throughput. We are evaluating the tradeoffs involved in this area.
- *Low-overhead computer communication protocol* implementation: We are looking at various communication protocols – from LAN and WAN communication to multiprocessor message passing – to see how they would be implemented with low overhead on Telegraphos, as discussed in section 3. Simultaneously, we are comparing and evaluating the Telegraphos communication mechanisms against the corresponding mechanisms in other multiprocessors.
- *Error recovery and synchronization* methods: In networks with reliable links, and which never drop packets, like the Telegraphos network (section 2.2), no packet acknowledgements and retransmissions are needed. Yet, some methods are necessary in order to handle the occasional loss of packets (due to the laws of probability in large systems), and in order to detect the arrival of all packets that must be delivered before a synchronization point in a parallel program is passed; we are working on such methods.
- *Eager sharing (updating)* (section 4) is a very important method in shared memory implementation. We are evaluating it and we are studying various details of its implementation.
- *Hardware monitoring* for assisting data and thread placement decisions (section 4.2): What is the most cost-effective and useful set of monitors and alarms?
- *Integration of shared memory and message passing:* Telegraphos uses the same basic mechanisms – a single address space and the remote write operation (plus a few variations of it) – to support both message passing (section 3) and shared memory (section 4). Does this show the path to a possible eventual unification of the two basic models of parallel computing – shared memory and message passing?

Acknowledgements

Telegraphos is being developed within ESPRIT project 6253 "Supercomuter Highly Parallel System" (*SHIPS*), funded by the European Union, through DG III of its Commission, HPCN Unit. We deeply appreciate this financial support, without which Telegraphos would have not existed.

Telegraphos is a collective effort: many people have contributed to the design of the architecture, the prototypes, and the systems software. The author wishes to acknowledge in particular Evangelos Markatos, Christos Nikolaou, Sarantos Kapidakis, and Peny Vatsolaki for valuable contributions to the architecture. Also, Richard Fortier, James Goodman, Robert Hyerle, Alasdair Rawsthorne, Marios Mavronicolas, and Falk Langhammer have helped in various ways, at various times. *Telegraphos I* is being implemented by Peny Vatsolaki, George Kalokerinos, Manolis Stratakis, George Papadourakis, and Miltos Grammatikakis. *Telegraphos II* is being designed by Apostolos Dollas, Chara Xanthaki, and Aristides Efthymiou. The Telegraphos Systems *Software* is being developed by Christos Nikolaou, Penelope Constanta-Fanouraki, Sarantos Kapidakis, Evangelos Markatos, George Dramitinos, Andreas Sakellaropoulos, Kosmas Papachristos, Marios Chryssopoulos, and Lambros Ganas. Dimitrios Serpanos helped with the bibliography. Useful comments on an early draft were provided by Dimitrios Serpanos, Costas Courcoubetis, Christos Nikolaou, Sarantos Kapidakis, and Miltos Grammatikakis. The help of all of them is greatly appreciated.

References

[CFFS93] D. Cohen, G. Finn, R. Felderman, A. DeSchon: "The Use of Message-Based Multicomputer Components to Construct Gigabit Networks", ACM SIGCOMM Computer Communication Review, vol. 23, no. 3, July 1993, pp. 32-44.

[CHKM93] R. Cypher, A. Ho, S. Konstantinidou, P. Messina: "Architectural requirements of Parallel Scientific Applications with Explicit Communication", Proceedings of the 20th Intnl. Symposium on Computer Architecture, ACM SIGARCH Vol. 21, No. 2, May 1993, pp. 2-13.

[C104] "IMS C104 Packet Routing Switch – Preliminary Datasheet", INMOS (member of SGS-Thomson Microelectronics), June 1993.

[DaSe87] W. Dally, C. Seitz: "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks", IEEE Trans. on Computers, Vol. C-36, No. 5, May 1987, pp. 547-553.

[Dally90] W. Dally: "Virtual-Channel Flow Control", Proceedings of the 17th Int. Symposium on Computer Architecture, May 1990, pp. 60-68.

[Dolp94] "DIS301 SBus-to-SCI Adapter User's Guide", Dolphin Interconnect Solutions AS, Oslo, Norway (e-mail "scimktg@dolphin.no"), April 1994.

[Dunb93] P. Dunbeck: "Digital's ATM Solutions", white paper distributed by DEC at Interop '93, Paris, November 1993.

[HePa90] J. Hennessy, D. Patterson: "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, 1990, ISBN 1-55860-069-8.

[HlKa88] M. Hluchyj, M. Karol: "Queueing in High-Performance Packet Switching", IEEE Journal on Selected Areas in Communications, Vol. SAC-6, No. 9, December 1988, pp. 1587-1597.

[HoLa93] M. Homewood, M. McLaren: "Meiko CS-2 Interconnect Elan - Elite Design", Meiko Ltd., 650 Aztec West, Bristol, UK, May 1993.

[Kate85] M. Katevenis: "Reduced Instruction Set Computer Architectures for VLSI", MIT Press, Cambridge, Mass., March 1985, ISBN 11103-9 KATCH.

[Kate87] M. Katevenis: "Fast Switching and Fair Control of Congested Flow in Broad-Band Networks", IEEE Journal on Selected Areas in Communications, Vol. SAC-5, No. 8, October 1987, pp. 1315-1326.

[KaSC91] M. Katevenis, S. Sidiropoulos, C. Courcoubetis: "Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip", IEEE Journal on Selected Areas in Communications, Vol. 9, No. 8, October 1991, pp. 1265-1279.

[Kung92] H. T. Kung: "Gigabit Local Area Networks: A Systems Perspective", IEEE Communications Magazine, Vol. 30, No. 4, April 1992, pp. 79-89.

[Leno90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy: "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor", Proceedings of the 17th Int. Symposium on Computer Architecture, May 1990, pp. 148-159.

[LiHu89] K. Li, P. Hudak: "Memory Coherence in Shared Virtual Memory Systems", ACM Transaction on Computer Systems, vol. 7, no. 4, Nov. 1989, pp. 321-359.

[Mapl92] C. Maples: "A High-Performance Memory-Based Interconnection System for Multicomputer Environments", Proceedings of the Supercomputing Conference, 1992, pp. 295-304.

[MaCh94a] E. Markatos, C. Chronaki: "Trace-Driven Simulation of Data-Alignment and Other Factors affecting Update and Invalidate Based Coherent Memory", Proceedings of the ACM Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '94), January 1994, pp. 44-52.P

[MaCh94b] E. Markatos, C. Chronaki: "The Use of Reference Counters in Update and Invalidate Based Coherent Memory", Proceedings of the Parallel Languages and Environments in Europe (PARLE 94) Conference, July 1994.

[SCI92] ANSI/IEEE Standard 1596-1992 "Scalable Coherent Interface", IEEE, 1992.

[SiDa79] J. Simon, A. Danet: "Controle des Ressources et Principes du Routage dans le Reseau Transpac", Proc. Int. Symp. on Flow Control in Comp. Networks, Versailles, France, Feb. 1979, pp.63-75; as reported in: L. Pouzin: "Methods, Tools, and Observations on Flow Control in Packet-Switched Data Networks", IEEE Transactions on Communications, Vol. COM-29, No. 4, April 1981, p. 422.

[Sites93] R. Sites: "Alpha AXP Architecture", Communications of the ACM, Vol. 36, No. 2, Feb. 1993, pp. 33-44.

[SwSm86] P. Sweazey, A. Smith: "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus", Proceedings of the 13th Int. Symposium on Computer Architecture, June 1986, pp. 414-423.

[Tyme81] L. Tymes: "Routing and Flow Control in TYMNET", IEEE Transactions on Communications, Vol. COM-29, No. 4, April 1981, pp. 392-398.

[WiMa89] L. Wittie, C. Maples: "Merlin: Massively Parallel Heterogeneous Computing", Proc. 1989 Int. Conf. on Parallel Processing, St. Charles, Illinois, Aug. 1989, pp. 142-150.

Index