TUMULT, the Twente University Multiprocessor

J. Scholten
P.G. Jansen

University of Twente
Department of Computer Science
P.O.B. 217
7500 AE Enschede
the Netherlands

Abstract

TUMULT stands for Twente University Multiprocessor. Its aim is the design and implementation of a modular extendible multiprocessor system. In the system up to 15 processing elements are connected through an interprocessor communication network. The basic method for the exchange of data is message passing. The hardware is controlled by a distributed realtime operating system, written in Modular Pascal and Modula-2. Each processing element may contain one or more processors (M680X0), memory boards and i/o interfaces, all connected through a VME bus, which acts as a local bus. The network offers a fully connected communication service with error detection and recovery. In the version realized, the transfer rate is 20 Mbyte/sec.

keywords: multiprocessor, MIMD, distributed operating system, realtime, interprocess communication, interprocessor communication.

1. Introduction

TUMULT is a research project of the University of Twente in conjunction with the Dr. Neher Laboratories (PTT) and Océ Nederland B.V. [8], [9], [10] Its aim is the design and implementation of a modular extendible multiprocessor system. The current system is of the MIMD type. [1] [5]. The architectural model of TUMULT, together with an overview of the system is presented in chapter 2. A description of both hardware and software is given in chapters 3 and 4. Current activities will be briefly discussed in chapter 5.

2. Architectural model

The system is a multiprocessor in which up to 15 processing elements (PE) are connected through an interprocessor communication network (IPCN) [FIGURE 1]. The PEs are loosely coupled and have no shared memory [21]. The basic method for the exchange of data is message passing.

Each PE consists of a processor (M680X0) with local memory, and may be extended with other processors, a floating-point processor or i/o. Because a standard type local bus is used (VME bus) extensions are realized with commercially available standard boards.

All nodes are situated in one or two 19" racks and are not geographically distributed. Because the distance between the nodes is very short, a fast, synchronous, parallel datapath can be used for interprocessor communication.

The PEs have two onboard serial ports, that are used for debugging purposes and for connecting terminals.

At this moment two versions exist of the IPCN:

- (1) the first (test) version is based on the VME bus and is still being used, mainly for writing applications and testing the operating system. The maximum number of PEs in this version is seven;
- (2) the second version has a ring network. Data is transferred through the ring in 16 bits parallel. The number of nodes possible in this configuration is 15.

Currently a third version of the IPCN is being developed, where 64 PEs can be connected.

The IPCN is completely transparent for the application: it doesn't matter which IPCN is used; the communication primitives offered by the system are the same for all IPCNs.

The hardware is controlled by a distributed realtime operating system, where each PE has its own multitasking subsystem [12], [10], [13].

3. Hardware

The hardware of TUMULT consists of up to 15 PEs, connected through a modular extendible network. This network is an uni-directional ring. [16], [19] The basic building block for the ring is the switching element (SE) [FIGURE 2], based on the concept of the DIMOND. [7]

The network interface (NI) connects the SE with the local bus of the PE. [FIGURE 3]

In the next paragraph the network is discussed in general terms. Then a description will be given of the main components of the communication hardware: the switching element and the network interface.

Network

The topology of the IPCN is a ring, which supports any-to-any communication. The TUMULT ring is characterized as an synchronous store-forward message switching network. For the lowlevel interprocessor communication three types of messages are used:

- (1) the data message,
- (2) the control message,
- (3) the exception message.

The first type transfers the actual data over the network. The second type controls the movement of data messages. The third type is used to raise an exception in another PE in case of malfunctioning hard- or software.

The width of all messages, without routing and control information is 16 bits. [FIGURE 4] These 16 bits of information are embedded in a frame together with 2 message type bits, 4 destination address bits, and 2 parity bits for error detection. If the address bits are all "1", the frame is marked empty.

The ring is constructed by concatenation of the desired number of switching elements. Each SE is capable of buffering one message (a slot). Messages on the ring are passed forward to the next SE at a puls of a common system clock.

When a node wants to send a message it waits for an empty slot. Then the slot is marked full by setting the destination address, the information bits are loaded, and the parity is generated. At the receiver side, the address in the passing slots is compared with the node address. If the addresses match, the information is removed from the slot, the slot is marked empty, and the parity is checked.

Because the slots are transferred at a rate of 10 MHz, and a node fills the slots with dma speed (2 - 4 MHz), one node is not able to fill all contiguous slots on the ring. Thus several pairs of nodes may communicate simultaneously through the ring.

To conclude, a summary of ring characteristics is given below:

network topology:	ring
control strategy:	circulating slots
access mechanism:	demand assignment
transmission mechanism:	synchronous store-forward
communication path:	message switching
	parallel

message format:

24 bits (16 information bits, 2 type bits, 4 address bits, 2 parity bits)

number of nodes:

max. 15

transmission rate:

10 M messages/sec.

Switching Element

The TUMULT ring is constructed with 2 x 2 switching elements [7]. These elements provide synchronous message switching, decentralized arbitration and error control. The SEs are interconnected by means of a parallel communication path. All SEs in a ring share a common clock, which determines the switching rate of messages around the ring.

Each SE has two input ports (i0 and i1) and two output ports (o0 and o1). [FIGURE 2] The output port o0 has a 24 bits wide register that holds one messageframe. At i0 a parity checker detects corrupted messages. Two parity bits are generated at i1. i0 and o0 are used for the interconnection of SEs, while i1 and o1 connect the SE with the NI.

The SE controls two external status signals, that are used by the NI:

shift_in: active when a message is available at o1,
shift_out: active when i0 holds an empty slot.

Three internal status signal are to be distinguished in the SE:

empty: all address bits of the slot at i0 are "1",
corrupt: a parity error is detected at i0,
select: the address of the slot at i0 matches the address of the SE.

The SE has three switching positions [FIGURE 5]:

pass_through (P): o0 is connected to i0,
accept (A): o0 is connected to i1,
dual (D): o0 is connected to i1
and
o1 is connected to i0.

Switching position can be described in term of the internal status signals:

P <-- (not corrupt) and (not empty) and (not select)
A <-- corrupt or empty
D <-- (not corrupt) and select

External status signals are directly derived from internal status signals:

shift_in <-- (not corrupt) and select
shift_out <-- corrupt or empty or select

Network Interface

Within TUMULT the network is separated from the node processor by means of a network interface. This interface isolates the processor from the functional and physical behaviour of the network, thus enabling both the processor and network to perform concurrently and asynchronously. In [FIGURE 6] a simplified model of the NI is shown. It is partitioned into three main modules:

- (1) the message receiver interface (MRI),
- (2) the message transmitter interface (MTI),
- (3) the dataflow control mechanism (DFC).

Each of the message types (data, control and exception) has its own input and output channel. Messages are buffered in the NI in fifos, which are used for synchronizing network and node processor. Omitting these fifos would result in severe degradation of network performance.

Message Receiver Interface: The MRI [FIGURE 7] receives messages from the SE and transfers them to the node processor. A typed message buffering is used for temporary storage. Depending on the type bits, the message is directed to one of four input channels by de demultiplexer. The reception of a control or exception message causes an interrupt to be generated at the node processor. Data messages are read from the fifo through a dma controller.

The demultiplexer routes messages, offered by the o1 output of the SE to the input channel. This is achieved by decoding the two type bits in the message into one of four store signals every time the shift_in signal is active. Three store signals (store_e, store_c and store_d) are used to buffer the 16 information bits of the message in respectively the exception, control or data input channel. The fourth signal (store_t) is

used by the dataflow control mechanism.

An input channel consists of a fifo memory capable of holding one or more words. The width of a channel is 16 bits. The depth depends on its function.

The fifo of the data input channel must have a fair capacity. In this case 64 words is considered sufficient.

The depth of the control input channel fifo is dictated by the amount of control messages a node processor may receive from other PEs. The minimal depth is 14, in which case each PE can receive one control message from every other PE. A more generous use of control messages is possible when a depth of 64 is chosen, thus enabling the PEs to send a maximum four (64 div 14) control messages per PE.

Exceptions are expected to be exceptional, so a single register will be sufficient for the exception input channel.

Figure 8: Message transmitter interface

Message Transmitter Interface: The message transmitter interface (MTI) [FIGURE 8] is used to send exception, control and data messages to other PEs. Furthermore, it sends token messages generated by de dataflow mechanism. Exception and control messages are written directly to the MTI by the processor, while a dma takes care of the transfer of data messages.

The multiplexer takes the contents of one of the four output channels and sends it to the SE. The data output channel is a single register, called the transit register. A dma cycle is requested when this register is empty, which means that the previous data message is accepted by the SE. Next to the transit register is the address register, in which the destination address is written once. As control and exception messages are not as frequent as data messages, no buffering is used in those output channels. They merely consist of a destination address register. After the address for those channels is written, the exception or control message is sent directly to the multiplexer by the processor. There address and information are merged and at an active shift_out signal shifted to the SE. The transfer of information and addresses to the multiplexer is controlled by the load_x and read_x signals. Load_x indicates that an output channel is allowed to transfer data, read_x indicates that the data to be written is present.

Dataflow Control: To avoid overflow of the input channels, a dataflow control mechanism is added in the NI [FIGURE 9]. This DFC is implemented for the data channel. Because exception messages are rare no flow control is needed there. Flow control for control messages is implemented in software. Software

implemented dataflow control would cause severe degradation of the system, because data messages are so frequent.

The hardware implemented DFC has two counters. One counter counts the number of data messages read by the node processor from the data input channel. This four bits counter is initialized with the number 15 and counts down. Every time it reaches zero a token message is sent to the sender of the data. Each token means that 16 words of information may be sent, because the receiver input channel has at least that amount of free space in its fifo.

At the sender side a second counter is active. This counter is initially loaded with the number 63, the number (-1) of free places in the input channel at the receiver. The counter is decremented every time a data message is sent. If the counter reaches zero, the dma transfer of data is stopped. Simultaneously the reception of a token increments the counter by 16.

Only in very rare occasions the counter reaches zero. The result is a continuous stream of data from sender to receiver at almost dma speed.

Figure 9: Dataflow control

4. Software

The TUMULT distributed realtime operating system is written in Modular Pascal [3] and Modula-2 [23]. Its layered structure is shown in [FIGURE 10].

The bottom four layers are equal for all PEs: **kernel** [12], **IPC** (interprocess communication), **file system** [18] and **LOM** (local manager) [2]. One PE has a **GLOM** (global manager), that coordinates the system. The upper layer is the application. Processes communicate via typed send- and receiveports with a message passing mechanism [14] [15]. The application is offered a number of primitives to open, close and connect to ports and links, where links are the communication channels.

Distributed Operating System

The distributed operating system is written in Modular Pascal [3], a language which allows the modular construction of programs. Modules can be compiled separately and are linked at runtime. Modular Pascal also offers the possibility to use a neat way of exception handling [4], which has been extended for distributed use in the TUMULT system.

A second release of the distributed operating system is written in Modula-2 [23].

The distributed operating system comprises the following layers:

kernel: a small and efficient realtime multitasking kernel [13] per processor is implemented, offering primitives such as process creation, termination and switching, process synchronization, interrupt handling, distributed exception handling, timer facilities, and memory allocation,

interprocess communication: the interprocess communication (IPC) layer [15] allows for dynamic creation of (virtual) communication links; these (virtual) links are used to transfer typed messages via send and receive ports,

distributed file system: files and devices can be distributed all over the nodes [20] [18]. The file system inherits its dynamic behavior from the IPC-primitives and therefore it tolerates the dynamic installation of devices and files,

local manager: the local manager receives, interprets and executes commands of the global manager, such as load, start or terminate a (sub) task, and it collects local status information,

global manager: the global manager interprets commands from terminal for the execution of (parallel) tasks. A task comprises one or more different subtasks. The GLOM distributes the subtasks over the nodes and orders the LOMs to execute them. The GLOM also collects global status information.

Interprocess Communication

Processes communicate via (typed) send- or receiveports by using a **message passing (MP)** mechanism, or a **remote procedure call (RPC)**. The MP mechanism can be chosen to be synchronous or asynchronous (buffered), the RPC is always synchronous.

The interprocess communication in the distributed operating system of TUMULT meets a number of primary requirements such as:

transparency of locality: the IPC offers transparency of locality in order to be able to reach any communication partner(s) without having knowledge of its residence. This is convenient in case of (dynamic) relocation of processes for load sharing purposes and/or for fault-tolerant system design.

reliability: messages are delivered correctly; in case of underflow (missing words), overflow (too much words), or parity faults, messages are not accepted by the receiver and re-sent by the system.

availability: due to the use of virtual circuit protocols, a sudden increase of communication requests will not cause severe system degradation (thrashing), which would result in unavailability of the system.

performance: the high performance IPC primitives, partly offered in hardware by the communication network, decrease the flow control overhead as much as possible thus increasing the efficiency of the available parallelism.

Ports are used to connect processes to a (virtual) communication link (identified by a so-called link name, which must be unique). We distinguish sendports (used by sender processes) and receiveports (used by receiver processes). A receiveport may have buffers.

The communication is based on a virtual circuit protocol (implying that a message is only sent if it can be received) on the hardware level as well as on the IPC level; no datagram services (implying that a message is always sent, regardless of the question whether it can be received) are used. A virtual circuit protocol may be less efficient than a datagram service; however, in our case a virtual circuit protocol is partly executed by dedicated hardware, which reduces considerably the efficiency drawback.

No dedicated broadcast facilities are implemented on hardware level. A functional (however not very fast) broadcast can easily be implemented by means of the existing communication primitives.

In TUMULT the following types of interprocess communication are supported:

parallel service requests (PSR): any process should be allowed to request a service, that resides at an arbitrary node. Since requesting processes may reside at an arbitrary node also, requests must be allowed to be sent in parallel to a service processor. The structure needed for this type of communication is 'uni-directional' and 'many-to-one' and is called a PSR link,

remote procedure call (RPC): a PSR link does not immediately support the RPC because no facilities are offered for the return of a reply. If replies are to be returned, separate return links must be used, one for each sender. This is neither convenient nor efficient and therefore a second structure, which is 'bi-directional' and 'many-to-one' is introduced. Such a structure is called an RPC link,

sequential data transfer (SDT): any process should be able to send a sequence of data records to an arbitrary receiving process that should receive these records in the order in which they were sent. The SDT link is derived from the PSR link.

TUMULT offers two basic communication structures: the PSR link and the RPC link. Links are virtual communication structures. At each node one or more processors may be connected to a link via the earlier mentioned ports. Links can be created, changed, and disposed of dynamically, according to the current needs of the system. PSR links may have buffer capacity in order to provide for buffered interprocess communication, they may also have a zero buffer capacity in which case the interprocess communication must proceed synchronously.

RPC links provide for synchronous communication only: there is no reason to buffer requests in a link buffer because a caller is waiting for an answer before it may continue. The requirements for sequential data transfer can be met in two ways: either by connecting only one sender and only one receiver to a unique communication link, or by offering the possibility of claiming the exclusive use as sender of a link, and the possibility of claiming the exclusive use as receiver of a link, thus creating a one-to-one noninterferable communication structure.

Parallel Service Request: Ports have to be declared in order to establish a connection between a process and a link. The declaration of a port is similar to the declaration of a file:

```
var p: port of <record type>;
```

Processes can be connected to a link via a port with the following procedures:

```
procedure consen (var sp: port; lname: string);
procedure conrec (var rp: port;
                 lname: string; bufsize: integer);
```

These procedures mark a port to be a sendport or a receiveport respectively. Exactly one receiveport and an undefined number of sendports may be connected to a link.

Figure 11: Ports and links

A port may be connected to a link only once, however, a connected port may be shared by other processes for which the port is in scope (processes may be declared at any lexical level; only those variables that are in the static environment of the process can be accessed).

'lname' is the name of a link, and it must be unique in the entire system. This name serves as a rendezvous key to connect sendports and receiveports to the same virtual link. In case of such a rendezvous component types of send - and receiveports must match. They determine the type of the messages which can be transferred via the link.

'bufsize' determines the size of the link buffer at the receiver node. It may be any positive number or to zero. In this way the number of receiver buffers can be defined by the caller of

'con_rec' (see also [11] and [22]). This offers the possibility to avoid system deadlock due to a lack of buffer capacity in circular communication structures.

An example of a many-to-one link configuration is given in [FIGURE 11].

Messages are sent or received by the following generic procedures respectively:

```
procedure send (sp, a);
procedure receive (rp, b);
```

'a' must be a variable of the same type as the component type of 'sp' and 'b' must be a variable of the same type as the component type of 'rp'.

Ports are disconnected from a link by:

```
procedure disconnect (p: port);
```

A port may only be disconnected by a process that connected it.

The port to be disconnected is either a sendport or a receiveport.

If a sendport has been disconnected, the particular reference to the link is made invalid; if a receiveport has been disconnected the complete link is made invalid and no operations on that link may be performed anymore except disconnect operations on send-ports. If no port is connected to a link anymore, the link is released.

Note that the amount of buffering in a link is defined by the user and can be equal to zero. In such a case communication between sender and receiver proceeds fully synchronously, as in CSP [6]: it is guaranteed that after a successful send action the message(=record) is received.

A link guarantees the following properties:

- (1) records are considered to be atomic in the sense that a record is received as it was sent and that no other records can intervene,
- (2) records sent by different senders via the same link are received in any order,
- (3) records sent by the same sender via one link are received in the order in which they are sent. Note that records sent by different senders may intervene,
- (4) if the number of link buffers is zero, the procedures 'send' and 'receive' are fully synchronized.

Remote Procedure Call: Within TUMULT a rudimentary type of RPC is offered: the caller sends a request to a callee and then waits for the reply; the callee receives the request, handles it in and sends the reply. The communication is performed synchronously; no buffering in the link is admitted. The same link is used to transfer requests as well as replies. Therefore, at the caller side as well as at the callee side, a port allows the transfer of two message types. They are declared in the following way:

```
var request: port of (<request record type>,
                    <reply record type>);
```

At the caller (client or request) side such a port can be connected to a RPC-link with identification "lname" by:

```
procedure con_client (var request: port;
                    const lname: string);
```

and at the callee (server or reply) side by:

```
procedure con_server (var report: port;
                     const lname: string);
```

At the caller side requests are sent from the variable "req" and replies are received in the variable "rep" by the following generic procedure:

```
procedure request_reply (report, req, rep);
```

Note that the first component type of "report" should match the type of the variable "req" and that the second component type must match "the type of "rep".

At the callee side the receiving of the request and the sending of the reply are separated in order to permit any form of programmable scheduling of the requested service: request can be queued in order to sustain the reply. A request and a unique identification of the sender are received by the following generic procedure:

```
procedure receive_req (report, req, sender_id)
```

in which "report" is a reply port, "req" is a variable of which the type must match the request type identification of the port, and "sender_id" is a variable of the type "sender_identification".

A reply is returned by the generic procedure:

```
procedure send_reply (sender_id, rep)
```

in which "sender_id" is a variable that contains a unique send_identification and "rep" is a variable that contains the reply.

The complete link will be claimed implicitly by a "send_request" action and other requesters will be blocked until a successful execution of "receive_reply" has occurred, after which the link will be released implicitly. Note furthermore that the communication pairs "send_request/receive_request" and "send_reply/receive_reply" are executed synchronously.

The ports of an RPC-link can be disconnected separately by the disconnect procedures.

Sequential Data Transfer: In order to exclude intervention of sequential data transfer between a sender and a receiver, a (unidirectional) PSR-link has to be claimed exclusively and released afterward. The following procedures are offered:

```
procedure claim (p: port);
procedure release (p: port);
```

There is exactly one sender that can successfully claim a link via its sendport for exclusive send activities and there is exactly one receiver that can successfully claim a link via its receiver port for exclusive receive activities.

The following procedures are used also for sequential data transfer:

```
procedure send_eos (ip: port);
function eos (op: port): boolean;
procedure receive_eos (rp: port);
```

The procedure "send_eos" is used to send an end-of-stream message to the receiver side in order to be used to indicate the end of a sequential data transfer.

The function "eos" is used to detect end-of-stream messages. The function blocks until a normal message or an end-of-stream message has been received; it yields "true" if the last

(first in the fifo queue) received message is an end-of-stream message and it yields "false" if the last received message is as a normal message.

The procedure "receive_eos" is used to consume the last received end-of-stream message.

For a more detailed description of the communication primitives, and for illustrations of their use we refer to [15].

5. Conclusion

In this paper an overview is given of the TUMULT multiprocessor; characteristic highlights of both hardware and software are presented.

Two prototypes of TUMULT are realized now: TUMULT-7, based on a VME bus as connection structure, and TUMULT-15 with a ring IPCN. These configurations are used for the development of the operating system. All software described in this paper, except for the remote procedure call, is implemented.

The following notes on system performance must be made. It is found that the communication hardware is in no way a bottleneck for the system. The bottleneck is found in the communication protocols overhead, which is approximately 4 milliseconds. The transfer time of a message is 1 microsecond per word.

Current research focuses on the design and implementation of TUMULT-64, based on the M68020 processor, and a bidirectional ring [17]. Communication will be handled by a communication processor, partly realized in VLSI. A SUN workstation, with a Modula-2 software development environment, is used as front-end processor for the system. Another topic is the research in the field of parallelisation of algorithms.

Applications for TUMULT are process control, prototyping, simulation, picture processing. It is intended to be used by the Dutch PTT for the recognition of handwritten numbers.

Acknowledgements

This project is a joined research effort of staffmembers of the University of Twente, Dr. Neher Labs, Océ Nederland B.V. and students. We thank all who contributed to the project and without whose help and support this project never would have been realized.

Literature

- [1] Anderson A.G., Jensen E.D.: "Computer interconnection structures: Taxonomy, characteristics and examples", ACM Computing Surveys, Dec. 1975
- [2] Brandsma E.: "The Global and Local manager of Tumult", M.Sc. Thesis, Dept. of Computer Science, Twente University of Technology, the Netherlands, Nov. 1985.
- [3] Bron C.W.: "Modular Pascal Language Definition", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-82-10, 1982.
- [4] Dijkstra E.J., Bron C.W.: "On the use of exception handling in Modular Pascal", Department of Computer Science, State University Groningen, the Netherlands, Sept. 1984.
- [5] Feng T.: "A survey of interconnection networks", Computer, Dec. 1981, IEEE Computer Society Press.
- [6] Hoare C.A.R.: "Communicating Sequential Processes", Comm. ACM, Vol.21, No.8, Aug. 1978, pp.666-677.
- [7] Jansen P.G., Kessels J.L.W.: "The DIMOND: A component for the modular construction of switching networks", IEEE Transactions on Computers, vol. C-29, no 10, Oct. 1980, p.884-889.
- [8] Jansen P.G.: "A survey of TUMULT, a real-time multiprocessor system", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-85-22, 1985.
- [9] Jansen P.G., Smit G.J.M., Scholten J.: "A survey of TUMULT, a real-time multi-processor system", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-86-2, 1986.
- [10] Jansen P.G., Smit G.J.M.: "TUMULT, a multi-processor system: its architecture and communication", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-87-3, 1987.
- [11] Kessels J.L.W.: "The Soma: A programming construct for distributed processing", IEEE Transactions on Software Engineering, Vol.SE-7, No.5, Sept. 1981.
- [12] Klaveringa E., Luttmer M.L.M.: "The TUMULT kernel", M.Sc. Thesis, Dept. of Computer Science, University of Twente, the Netherlands, March 1986.
- [13] Luttmer M.L.M., Jansen P.G.: "The TUMULT kernel", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-87-15, March 1987.
- [14] Ribbers H.: "The design and implementation of a Link Manager for a multiprocessor system", M.Sc. Thesis, Dept. of Computer Science, University of Twente, the Netherlands, Aug. 1985.
- [15] Ribbers H., Jansen P.G.: "Communication facilities for TUMULT", Dept. of Computer Science, Twente University of Technology, the Netherlands, int. rep. no. INF-87-10, 1987, Febr. 1987.
- [16] Scholten J., Smit G.J.M., Van 't Hoff E.L.: "Realization of an interconnection network for TUMULT", Euromicro Symposium, Brussels, Sept. 1985, reprints, p327-334.
- [17] Scholten J., Smit G.J.M., Hofstede J.: "An architecture for TUMULT based on a serial data link", Euromicro '87, Portsmouth, proceedings, p613-620, 1987.
- [18] Sijbers A.: "A distributed File System for TUMULT", M.Sc. Thesis, Dept. of Computer Science, Twente University of Technology, the Netherlands, 1987.
- [19] Smit G.J.M., Scholten J.: "An interconnection network for TUMULT", Proceedings NGI/SION 1985, p209-221.
- [20] Van Langen P.J.M., Sijbers A.L.M.T.: "The distributed filesystem in TUMULT", Dept. of Computer Science, University of Twente, the Netherlands, 1987.
- [21] Vuurboom R.: "Communication in a loosely-coupled multiprocessor system: its architecture and implementation", M.Sc. Thesis, Dept. of Computer Science, University of Twente, the Netherlands, 1984.
- [22] Waumans B.L.A.: "Software aspects of the PHIDIAS system", Philips Technical Review, Vol.40, No.8/9, 1982.
- [23] Wirth N.: "Programming in Modula-2", Third corrected edition, Springer Verlag, 1985.

Glossary

DFC	DataFlow Control
GLOM	GLObal Manager
IPC	InterProcessor Communication
IPCN	InterProcessor Communication Network
LOM	LOcal Manager
MP	Message Passing
MRI	Message Receiver Interface
MTI	Message Transmitter Interface
NI	Network Interface
PE	Processing Element
PSR	Parallel Service Request
RPC	Remote Procedure Call
SDT	Sequential Data Transfer
SE	Switching Element
TUMULT	Twente University MULTiprocessor