CONSTRUCTING REPLICATED SYSTEMS USING PROCESSORS WITH POINT TO POINT COMMUNICATION LINKS

by

Paul D. Ezhilchelvan, Santosh K. Shrivastava and Alan Tully

Computing Laboratory, University of Newcastle Upon Tyne, England, U.K.

Abstract

Replicated processing with majority voting is a well known method of achieving fault tolerance. We consider the problem of constructing a distributed system composed of an arbitrarily large number of N-modular redundant (NMR) nodes, where each node itself is composed of N, $N=2m+1$ and $m \geq 1$, processing and voting elements. Advanced microprocessors, such as Inmos Transputers, provide fast serial communication links for inter-processor communication, making it possible to construct large networks of processors. We describe how replicated processing with majority voting can be achieved for such processor networks. This paper will present the overall systems architecture, including voting and NMR synchronization algorithms specially developed to exploit fast point to point communication facilities.

Keywords: Replicated processing, majority voting, N-modular redundancy, sequencing algorithm, fault tolerance

1. Introduction

We consider the problem of making a system of concurrent processes tolerant to a bounded number of processor failures. Given a non-redundant system of C ($C \geq 1$) concurrent processes partitioned to run on P ($P \leq C$) number of processors, we address the problem of constructing a *voted replicated system* of N*C processes ($N=2m+1$, $m \geq 1$) partitioned to run on N*P processors and capable of tolerating up to P*m processor failures. Each process $c \in C$ is replaced by a group of processes with N members, and each processor $p \in P$ is replaced by a group of N processors. We assume that processes interact by message passing. This requires that all message interactions must be voted upon in the replicated system. It is our objective to achieve *replication transparency* such that any problems posed by replication and voting are hidden from the application programmer who is then only concerned with the development of a non-redundant system.

Replicated processing with majority voting -N-modular redundant (NMR) processing- is a well known method of achieving fault tolerance. The problem of applying this method to distributed and multicomputer systems has received much attention [1-5]. Many fault tolerant distributed systems have been implemented under a rather restricted fault assumption, which is that processors fail "cleanly" by just stopping [eg. 6]. Such an assumption is hard to justify in computer systems intended for mission and life critical applications where failure probabilities in the range 10^{-6} to 10^{-10} per hour are often specified [3,4]. It is then necessary to design and implement such systems under a highly unrestricted fault assumption, namely, that a failed processor can behave in an arbitrary manner (in the literature this failure mode is often referred to as the *Byzantine failure* mode [7]). While certainly not common, experience has shown that Byzantine failures cannot be ruled out in the design of fault tolerant systems [3-5]. NMR processing, whereby outputs from faulty processors can be masked by voting, provides a practical means of constructing systems capable of tolerating Byzantine processor failures. In this paper we develop a specific architecture necessary for supporting replicated processing. This architecture exploits the following property that we assume for all processors: each processor has a fixed number of communication links through which processes executing on that processor may send or receive messages from processes of other connected processors. In the following we shall present a processor interconnection and communication scheme together with voting and sequencing algorithms necessary for replicated processing.

The overall architecture that we present here is of practical importance. This is borne out of the fact that some current microprocessors such as the Inmos Transputer [8] provide just the kind of communication facilities assumed here (a Transputer has 4 bi-directional 10Mbit/sec serial communication links). Using such links, large multi-Transputer networks can be built for specific applications. A number of these applications make use of *Transputer farms* (either single or two dimensional Transputer arrays) for parallel processing. The architecture developed here can be used for such applications. NMR processor networks have also been proposed for realtime control applications such as railway signalling [9]. Thus there is every reason to develop an optimized communications architecture for such systems. From now on we will assume the degree of replication to be three giving us the well known Triple Modular Redundant (TMR) system. The paper is structured as follows. The second section develops a model for replicated processing and describes the voting and sequencing requirements. Section three presents a processor interconnection scheme which meets the TMR criterion of tolerating one processor failure per TMR node. Section four presents the algorithm for voting and sequencing. In section five the design presented here is compared and evaluated with other approaches reported in the literature. Conclusions from our work are presented in section six.

2. Replicated Processing Model

We assume that application programs can be mapped on to a number of processes that interact via messages. *Communicating processes have bi-directional links* between them. Figure 1a shows a system of five concurrent processes with six links. A link connecting two processes has the property that the messages sent by one process are received by the other process uncorrupted and in the sent order. We also assume that if a process with multiple links (such as c_2 in Figure 1a) simultaneously receives messages on those links then these messages are chosen non-deterministically for processing. Message selection is however assumed to be *fair*, that is, the process will eventually select a message present on a link. We assume that such a system of processes can be configured to run on a given set of processors. Each processor is assumed to possess a fixed number of bi-directional communication links. For example, Figure 1b shows a particular configuration for three processors with c_1 and c_2 mapped onto processor p_1, c_3 mapped onto p_2 and c_4 and c_5 mapped onto p_3. Interprocess links l_i are also mapped onto physical links s_j connecting the host processors (eg. l_2, l_3 mapped onto s_1). A physical link connecting two processors, such as s_1, is composed of a bi-directional link from p_1 and a similar link from p_2 connected electrically to form a single bi-directional link. The model presented here is sufficiently general in that other models, such as clients and servers interacting through remote procedure calls, or objects communicating by messages can be seen as special cases. It is also worth noting that processors such as Transputers directly support a processing model very similar to the one presented here via the Occam programming language [10].

A processor with its links will be treated as a single entity, so a processor with either faulty links or a faulty processing unit or both will be treated as a faulty processor. A failed processor may behave in an arbitrary manner and hence, processes of a failed processor may behave in an arbitrary manner. In the replicated version of the system, each process c_i ($1 \le i \le 5$ in Figure 1b) will be replaced by a process triad C_i, such that $C_i \equiv \{C_{i1}, C_{i2}, C_{i3}\}$ and each processor p_j ($1 \le j \le 3$ in Figure 1b) will be replaced by a processor triad P_j such that $P_j \equiv \{P_{j1}, P_{j2}, P_{j3}\}$ with C_{i1} mapped onto P_{j1}, C_{i2} onto P_{j2} and C_{i3} onto P_{j3}. Computations performed by a process are assumed to be *deterministic*. In particular this means that if all non-faulty processes of a triad have identical initial states and process identical messages in an identical order, then identical output messages in an identical order will be produced. To simplify subsequent discussions, it will be assumed that the function of a computational process, such as C_{ij}, is simply to remove the selected message from the link and output a message after some processing. Assuming that at most one processor in each triad may fail, at most one process in each triad may be faulty. A non-faulty process must reject inputs from a faulty process; this is achieved by majority voting as shown in Figure 2 which depicts the voters for the jth process triad of C_1 (The replicated version of process c_1 in Figure 1a).

In Figure 2, if C_{1j} receives two voted messages simultaneously then one of them will be chosen non-deterministically for processing. It is however necessary for all the non-faulty members of a process triad to make an identical selection. This is difficult to achieve even if it is assumed that non-deterministic choice is replaced by a fixed priority selection criterion, since voted messages need not arrive at the same time at all processes of the triad. We will convert the problem of identical message selection to

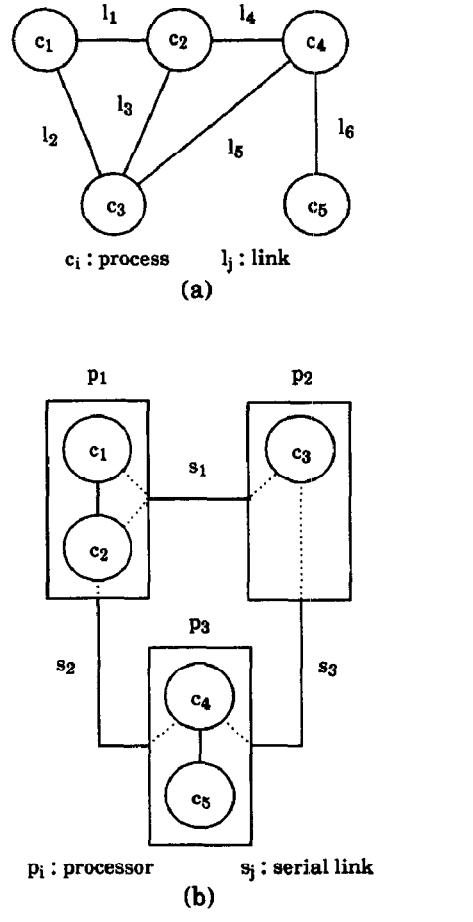


Figure 1: Process-Processor Mapping

that of *identical message ordering* by serializing the inputs to a process as shown in Figure 3. All voted messages are stored for processing in the voted message pool (VMP). The *order processes* of a triad cooperate to select a voted message for processing using a protocol which will be discussed later. The selected message is inserted in the voted message queues (VMQs). Every process C_{ij} has a VMQ_{ij} associated with it. Process C_{ij} picks up the message at the head of its VMQ and processes it. The results are

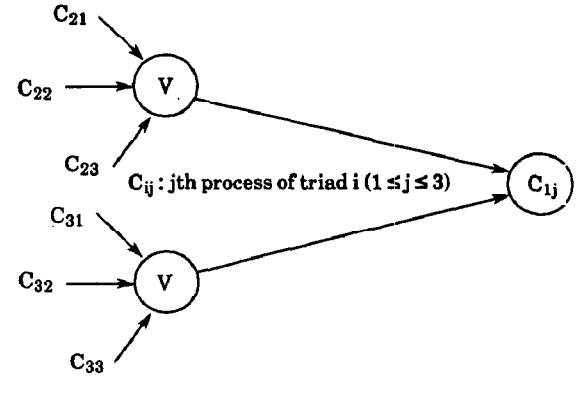


Figure 2: Voting of incoming messages

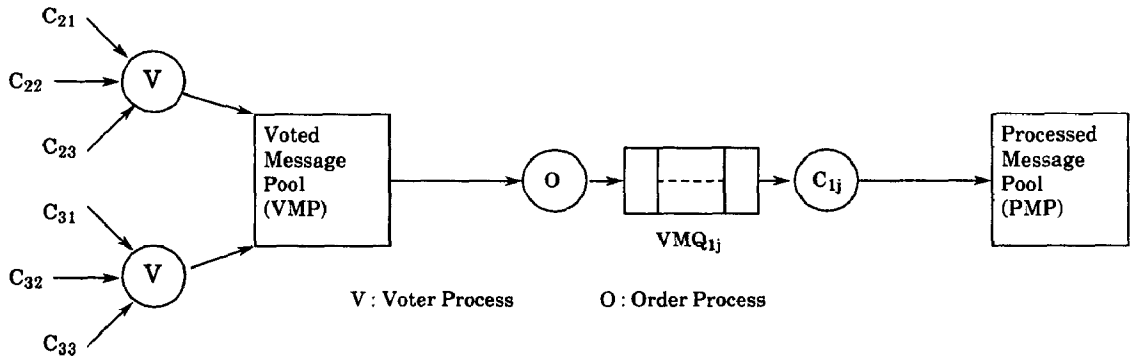


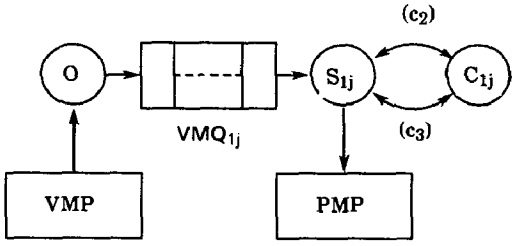
Figure 3: Processing of Voted Messages

stored in the processed message pool (PMP) for transmission to the relevant triad. We now require the following *sequencing condition* to be satisfied by all non-faulty processes of a triad :

SEQ: All the non-faulty computational processes of a triad process voted messages in an identical order.

The order processes ensure that SEQ is met by delivering voted messages to the VMQs in an identical order.

Transforming a system such as shown in Figure 2 to it's equivalent form shown in Figure 3 has several advantages. Communications software can be developed for maintaining the message pools (Figure 3) which can be shared between all the processes mapped onto that processor. Similarly, a single order process per processor can be utilized for delivering messages to all the VMQs of the processes of that processor. Finally, a single voter process per processor can be utilized for voting messages. Voting can be carried out either on *incoming messages* (at the receiver triad) or on *outgoing messages* (at the sender triad, in which case receiver triads receive only voted messages). We will assume that the voter of a processor votes all outgoing messages (the reason for this choice will be explained subsequently after processor interconnections have been discussed). A shortcoming of the scheme shown in Figure 3 is that process C_{1j} has only one incoming link (from VMQ_{1j}) and one outgoing link (to PMP) and hence it no longer communicates like its unreplicated counterpart c_1 which has bidirectional links with c_2 and c_3. This shortcoming can readily be overcome by introducing a *stub process* S_{1j} into the host processor of C_{1j} to emulate the two original links to c_2 and c_3 as illustrated in Figure 4. Process S_{1j} interfaces to VMQ_{1j} and the PMP: it picks up a message



S_{1j} : stub process to emulate c_2 and c_3

C_{1j} : jth replicated version of c_1

Figure 4: A Stub Process

from VMQ_{1j} and sends it to C_{1j}; a message from C_{1j} on the other hand is deposited in the PMP for voting. It can be seen that stub processes are similar to client and server stubs employed in remote procedure call systems [11]. In the subsequent discussions, for the sake of simplicity, we will assign to C_{ij} the role played by the stub process S_{ij} thereby supposing that process C_{ij} interacts directly with VMQ_{ij} and PMP as shown in Figure 3.

To summarize: a processor maintains two pools: VMP and PMP. Each process C_{ij} on that processor processes messages on its VMQ_{ij}. The order process of the processor selects messages from the VMP and inserts them in the appropriate VMQs. The voter process of the processor votes messages from the PMP before transmitting them.

3. Processor Interconnection and Message Diffusion

We will assume that a processor has four bidirectional links. Four links are enough to construct a pipeline (or a ring) of processor triads as shown in Figure 5.

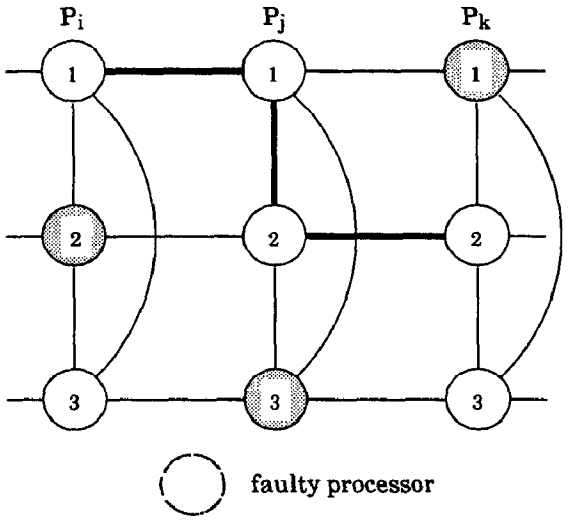


Figure 5: Processor Interconnections

The pipelined nature of interconnection means that a message intended from one processor to some other processor may have to be relayed by intermediate processors. From a TMR system we require the capability of masking at most one processor failure per triad. Under

such a failure assumption, the pipelined system possesses the following two properties:

(i) Any non-faulty processor of a triad is directly connected to all other non-faulty processors of the same triad.

(ii) From any non-faulty processor in a triad P_i, there is a non-faulty path connecting that processor to any non-faulty processor of triad P_k. A path between any two non-faulty processors is non-faulty if all the intermediate processors in the path are non-faulty (for example, the path between P_{i1} and P_{k2} drawn in bold lines in Figure 5 is non-faulty).

We will refer to a link connecting processors of a triad as *internal* and a link connecting processors of two adjacent triads as *external*. All the processors of the system as well as processor and process triads will be assumed to possess unique names. We will assume the existence of some name mapping scheme for process triads such that a sender or relayer can determine whether the destination triad is on its 'left' or 'right'. In addition, it will be assumed that a *message contains an ordernumber (a form of sequence* number, see section 4). The following message handling primitives will be assumed (to be provided by the communications software of a processor):

(i) *send(message)* : The message intended for the named destination process triad, where the destination process triad is not on the same triad as the sender, is sent on the appropriate external link.

(ii) *diffuse_internal(message)* : The message is sent over all the internal links of a processor.

(iii) *diffuse_external(message)* : This operation is invoked by a relaying processor if it receives a double signed message on one of its links. If a message is received on an external link, then the processor sends the received message on all internal links as well as the external link which is towards the direction of the destination. If a message is received on an internal link, then it is only forwarded outwards via the appropriate external link. Optimizations are possible, since a processor need not relay a message already relayed.

(iv) *receive(message)* : Receive a message that has arrived on any link of the processor.

It is necessary for a receiver to be able to detect messages which have arrived altered. This is particularly important within the context of a pipelined architecture where messages can be relayed. For this reason, a message must contain enough redundancy. If it is assumed that faulty processors can behave in an arbitrary manner, then sophisticated authentication techniques are required to detect message corruptions with high probability [12]. We assume that each processor has a mechanism to generate a unique unforgeable signature for a given message and further that each processor has an *authentication function* for verifying the authenticity of a signature. Thus if a non-faulty processor sends a message with its signature to some other non-faulty processor, any corruption of this message by a relaying processor can be detected by the receiver by authenticating the signature associated with the message. These assumptions about signatures are identical to those made by other researchers [7,13]. The *send* primitive and the *diffuse_internal* primitive automatically sign the message to be sent On the other hand, the *diffuse_external* primitive does not sign the message (since this operation is invoked for relaying a message).

If more than four links per processor are available, then structures other than a pipeline can be formed. For example, Figure 6 shows a triplicated grid structure which requires six links per processor. If necessary, a six link "processor" can be built using two four link processors as shown in the figure.

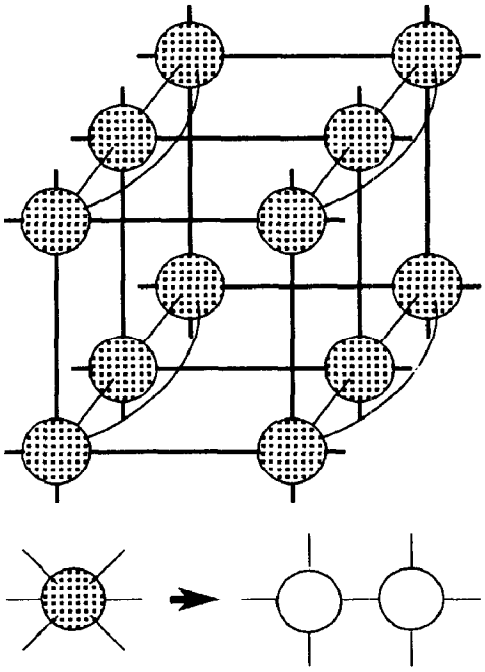


Figure 6: A Triplicated Grid

The above message handling facilities are used by voting and ordering processes as described in the next section.

4. Voting and Ordering

Recall that there is one voter process in every processor and that messages produced by a member process of a triad are voted before being sent to the destination. Assume that a process triad C_m mapped onto processor triad P_i (Figure 5), is sending voted messages to process triad C_n on processor triad P_k. The voting of messages is achieved as follows: any message produced by a member process of C_m is diffused internally to its neighbours. When a member processor of triad P_i accumulates at least two messages which agree, it sends this message on one of its external links depending on the location of the destination triad (signed messages are authenticated as we will see shortly). Figure 7 shows the flow of messages from source triad to destination triad.

For the purpose of sending and receiving voted messages, a processor maintains message pools VMP, PMP and two additional pools:

(i) *Voted Message Pool (VMP)* : Contains voted messages intended for members of process triads (C_{ij}'s) mapped onto this processor.

(ii) *Processed Message Pool (PMP)* : Contains processed messages deposited by C_{ij}s These messages must be voted before transmission

(iii) *Received Message Pool (RMP)* : Contains signed messages that are received for voting and found to be authentic.

(iv) *Candidate Message Pool (CMP)* : Contains unsigned messages waiting for a signed message of identical contents to arrive at RMP.

Two operations are defined on these pools:

remove(pl,m): a message is removed from pool pl and assigned to m.

deposit(pl,m): message m is deposited in pl, if m is not already present in pl.

As stated before, each process C_{ij} has a VMQ containing messages to be processed and each message contains an *ordernumber*. Messages in a VMQ are queued such that if m_1 and m_2 are in the VMQ and m_1 is before m_2 then $m_1.ordernumber < m_2.ordernumber$. This ordernumber is generated by the *order* process as it deposits a voted message in a given VMQ. As we shall see, the order process composes such a number by taking the current local clock time and concatenating it with the host triad identifier. The ordernumber of a message is a convenient means of determining its "age" within a processor and can be utilized for detecting unwanted messages (see process *flush* below). Messages have the property that their ordernumbers are unique within a given non-faulty processor. The function of a process such as C_{ij} is to process messages in its VMQ:

A result message deposited in the PMP retains its original ordernumber (it is not modified by C_{ij}).

The voter process is composed of four concurrent processes: *sender, receiver, majority* and *flush*. In the algorithms we will make use of the following notation:

m.dest : destination processor triad of message m.
m.destproc : destination process triad of message m.
m.ordernumber : ordernumber of message m.
ordernumber.clockvalue : clock value component of ordernumber

The *sender* process diffuses messages internally for voting. The receiver process collects all the messages sent by member triad processors for voting in the pool RMP. All voted messages contain two signatures from two members of the sending triad. If the received message is signed twice and is not meant for the receiver then the message is relayed by diffusing it using the *diffuse_external* procedure, otherwise the message is deposited, if it is authentic, in the VMP. The majority process tries to vote messages from pools CMP and RMP. If a majority can be formed, the message from RMP is countersigned and the double signed message is sent to the destination if the destination is remote, otherwise the countersigned message is and deposited in the VMP. From then on, double signed local messages in the VMP are treated like any other voted messages received from other triads. In addition to the three processes discussed above, it will be necessary to have another process -*flush*- which occasionally runs to flush out unwanted messages from the RMP (messages are unwanted if their replicas have already been voted upon; in addition, those messages, sent by faulty- possibly malicious- processors, that will never find a matching message from CMP are also unwanted). The constant D_{max} should be based on an extremely generous estimation of the time taken for any message to progress from the instant it is deposited in a VMQ to CMP

```
process C_ij :
    cycle
        pick up the message at the head of VMQ_ij
        process the message
        deposit the result message in PMP
    end
end C_ij

process voter :
    { process sender :
        var m:message
        cycle
            remove(PMP,m)
            deposit(CMP,m)
            diffuse_internal(m)
                /*signed m is sent to neighbours */
        end
    end sender
    //
    process receiver :
        var m:message; me:host_triad_identifier
        cycle
            receive(m)
            if m is signed once & authentic →
                deposit(RMP,m)
            □m is signed once & not authentic → discard
            □m is signed twice&m.dest ≠ me →
                diffuse_external(m)
            □m is signed twice & m.dest = me & authentic
                → deposit(VMP,m)
            □m is signed twice & m.dest = me
                & not authentic →
                discard /* corrupted message */
            fi
        end
    end receiver
    //
    process majority :
        var m:message; me: host_triad_identifier
        cycle
            m: = a message from RMP identical to a
            message from CMP
            /* such a pair is removed from these pools */
            if m.dest ≠ me → send(m)
            /* double signed voted message
            is sent to its destination */
            □m.dest = me → countersign and
            deposit(VMP,m) /* local message */
            fi
        end
    end majority
    //
    process flush:
        var T:timestamp
        cycle
            T: = local_clock_time - D_max
            remove any messages from RMP with
                ordernumber.clockvalue ≤ T
            remove any messages from RMP with
                ordernumber.clockvalue >
            local_clock_time
            wait for a while
        end
    end flush
    }
end voter
```

Recall that the ordernumber.clockvalue of a message records the local clock time when that message was put in the VMQ. Thus any message with ordernumber.clockvalue less than the current local clock time minus D_{max} can be deemed to be unwanted. Similarly, any message with ordernumber.clockvalue greater than the current local clock time must also be treated as unwanted (this is because a message must take a finite amount of time to progress from a VMQ to the CMP). The flush process performs an important function as it prevents an overflow of the RMP; in its absence, a faulty processor of a triad can cause non-faulty member processors to fail by creating overflows. It is worth noting that since inter-triad messages are voted and double signed at the senders and accepted only if authenticated at the receivers, a single failed processor of a triad cannot successfully send arbitrary messages to remote triads; so flush processes are required for intra-triad traffic only. The advantages of voting at sender triads can be appreciated given the processor connection schemes shown in Figures 5 and 6: inter-triad message traffic is cut down by a factor of three.

The function of the *order* process is to pick up a message from the VMP and assign it a *new ordernumber* and place it in the appropriate VMQ. The order process of a non-faulty processor must assign ordernumbers that are identical to those assigned by any other order process of a non-faulty member triad processor. We assume the existence of an *atomic broadcast* mechanism [13] between the member processors of a triad. The clocks of non-faulty processors of a triad are also assumed to be synchronized such that the measurable difference between readings of clocks at any instant is bounded by a known constant.

We require the properties of *atomicity, validity, termination* and *order* from the atomic broadcast mechanism. When a sender broadcasts a message m at local clock time T then:

(i) m is delivered to either all non-faulty receivers or to none of them (*atomicity*);

(ii) if the sender is non-faulty then m is delivered to all non-faulty receivers (*validity*);

(iii) m is delivered to all non-faulty receivers at their local clock time $T + \Delta$ (*termination*); and

(iv) all the messages broadcast by non-faulty senders are delivered to non-faulty receivers in an identical order (*order*);

where Δ is some known bounded quantity.

Two primitive operations for atomic broadcasts will be assumed:

(i) *Acast(msg)*: message "msg" is broadcast to member processors of the triad, including itself.

(ii) *Areceive(msg)*: message "msg" is received.

The *order* process is composed of three processes: *broadcaster, receiver* and *sequencer*. A queue of messages named *broadcast message queue* (BMQ) is maintained by the order process. Two standard queue operations -*put* and *get*- are available on a BMQ.

The broadcaster process ensures that every voted (double signed) message is made available to other member triads. Referring to figure 5, it can be seen that only the second processor of P_k will receive a voted message from P_i; local broadcast of this message will thus ensure that the message gets distributed. In general then, a processor can receive at most two extra copies of a voted message through broadcasts. The receiver process receives atomically broadcasted messages and deposits them in the BMQ. Because of the use of the atomic broadcast facility, the BMQ of a non-faulty processor possesses the following *stability* property: at any local clock time *t*, the state of the BMQ will be identical to the state of the BMQ of any other non-faulty member processor triad at its local clock time *t*. This property of BMQs is exploited for sequencing and generating new ordernumbers for messages by the sequencer processes. Assume that the initial start time (some value greater than the current local clock time) and the constant ∂ are the same for all the sequencer processes of a triad. Then all the non-faulty sequencer processes of a triad enter the loop at identical local clock times and then

```
process order:
  {process broadcaster:
     var m:message
     cycle
        remove(VMP, m)
        Acast(m)
     end
  end broadcaster
  //
  process receiver:
     var m:message
     cycle
        Areceive(m)
        if m is signed twice &
        authentic → put(BMQ,m)
           /*put in BMQ*/
        □m is not signed twice & authentic → discard
           /*spurious message*/
        □m is signed twice & not authentic → discard
           /*spurious message*/
        fi
     end
  end receiver
  //
  process sequencer:
     var m:message; T: timestamp;
     var i:process_triad_identifier
     T:=initial start time
     wait until local_clock_time=T
     cycle
        get(BMQ,m)
           /*remove the message at the head of BMQ
           */
        if m ≠ null → i:=m.destproc
           /*get the identifier of source process */
           if Delivered_VMQ(m,i) → discard
              /*replica */
           □ not Delivered_VMQ(m,i) →
              m.ordernumber:=T concatenate
              host_triad_identifier
              strip off double signatures from m
              deposit in the VMQ of process i
           fi
        □m=null → skip
        fi
        T:=T+∂
        wait until local_clock_time=T
     end
  end sequencer
  }
end order
```

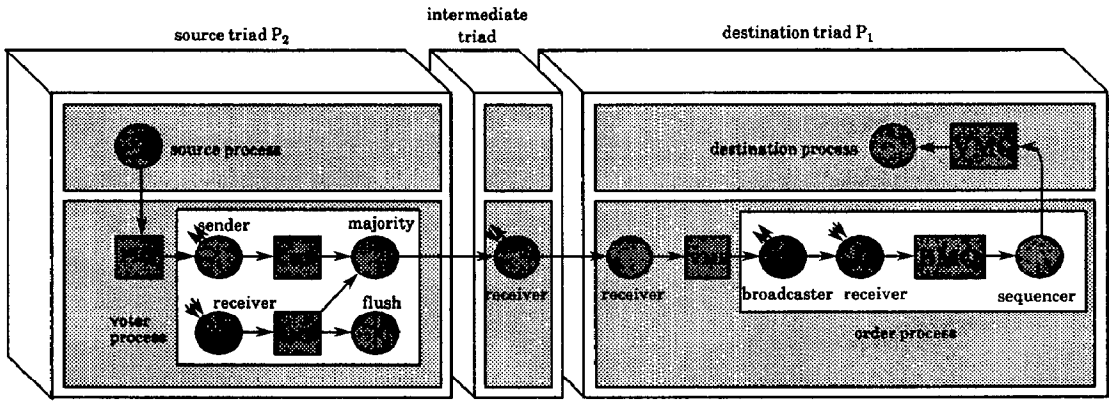


Figure 7: Message Flow

periodically remove identical messages from BMQs. The period ∂ must be chosen to be greater than the maximum processing time within the loop. Since a BMQ can contain replicas of a message, a function *Delivered_VMQ(m,i)* is assumed to exist to determine whether m is a replica of a message that has already been delivered to the VMQ of process i. Clearly, a sequencer process will have to maintain some state information about the messages delivered to the various VMQs, but these details have been omitted here for the sake of simplicity. Some optimizations to the scheme presented here are possible and briefly mentioned here. A protocol can be designed to reduce the number of broadcasts by exploiting the fact that a processor need not broadcast a message if it has already received a similar message from a broadcast from some other member processor. Also it is possible to design an integrated method for broadcasting and synchronization of clocks to reduce the number of messages needed just for clock synchronization [14]. Finally, voted messages whose destinations are local, need not be double signed, and can be treated specially, rather than uniformally as done here, to save processing time.

5. Design Evaluation

Our claim that the architecture presented here is suitable for point to point link based communication systems is based on the observation that while communication between directly connected processors can be fast, communication which involves intermediate processors for relaying can be considerably slower. Thus a good system design should minimize the number of inter-triad messages and eliminate altogether any need for complex protocols between any two processor triads. We achieve this by voting at the source triad, so at most three voted messages are sent by a triad, and by requiring the facilities of clock synchronization and atomic broadcast only for individual triads.

SIFT [3] and DELTA-4 [5] are two well known examples of systems capable of supporting NMR processing; both rely on bus or LAN based communications. SIFT employs a global clock synchronization facility for scheduling of *periodic* tasks only, while DELTA-4 employs special purpose LAN hardware to support atomic broadcasts between any set of communicating entities. Neither of these approaches is suitable to the kind of systems considered here.

There are essentially two classes of solutions to the problem of meeting the sequencing condition. One class of solutions makes use of Byzantine agreement protocols where no attempt is made to detect faulty processors while the other class of solutions make use of timeouts for detecting possibly faulty processors. The solution adopted in this paper falls in the first class since atomic broadcast algorithms require Byzantine agreement (recall that we are assuming arbitrary behaviour by failed processors). It is possible to perform Byzantine agreement at a higher level than proposed here, for example, by performing agreement on VMQs, as discussed in [15]. In this scheme, voted messages are inserted directly in VMQs, and order processes *periodically* execute a Byzantine agreement algorithm called *join* to agree upon a common ordering for messages present in all VMQs. This solution can be attractive if a reasonable period for executing the algorithm can be determined. There is however no simple means of determining this period. Our solution avoids this problem by making use of atomic broadcast *before* entering messages in VMQs.

A very interesting scheme - which does not make use of Byzantine agreement and thus falls in the second class of solutions - has been proposed in [16]. This scheme optimizes the number of messages that need be exchanged between process triads for voting and sequencing. Here processes at sender and receiver triads are partitioned into primary and secondary groups and a clever *distance voting* scheme is used for sequencing. This scheme appears unattractive in the system proposed here for two reasons. Firstly, it requires accurate assessment of timeouts for detecting absence of messages - this is not easy if messages have to be relayed through intermediate processors; and secondly, the scheme is very much 'process triad based', such that there does not appear to be any means of delegating the task of voting and ordering to 'system processes' that can be shared by all the processes of a processor. This makes the task of achieving replication transparency much harder. A scheme which also makes use of primary and secondary groups has been proposed in [17]; however it assumes a restricted 'fail stop' failure model for processors.

Another approach to solving the sequencing problem is to use another voter between a VMQ and C_{ij} [18]. If such an input voter detects a disagreement or cannot form the majority then this could either be due to a sequencing failure (all three messages being voted are different) or due to processor failures or both. Thus, it is necessary to be able to distinguish sequence failures from processor failures. The paper [18] describes a means of achieving this. A possible shortcoming of this approach is that it makes voting algorithms rather complex. We believe that if atomic broadcast among processor triads can be achieved

cheaply then the scheme presented here is more attractive. Input voting has also been suggested in the VOTRICS system [9]. This system employs a processor interconnection scheme which is similar to ours. The description of input voting presented there is however not in sufficient detail to enable us to evaluate the approach.

Finally, we would like to mention the scheme described in [19] where sequence failures are detected as *exceptions* by voters and backward error recovery is performed by processes causing the sequencing failure. In the present paper we have assumed that backward error recovery facility is not available, thus ruling out this solution.

As stated before, the atomic broadcast mechanism requires the properties of termination and atomicity. To achieve termination, it is necessary to have the clocks of all non-faulty processors of a triad to be synchronized such that the measurable difference between the readings of clocks at any instant is bounded by a known constant (say ϵ). To achieve atomicity, each processor is required to relay the message received as a result of a broadcast to both of its neighbours. Let d be the maximum unpredictable message transmission delay, then the lower bound for constant Δ of atomic broadcast can be shown to be equal to $2(d+\epsilon)$ [7]. Clock synchronization error ϵ will be of the order of d [20] thus giving the value of Δ to be of the order of a few d. Since processors of a triad are directly connected to each other, d can be made quite small by employing special low level protocols. With Transputers for example, d for short messages can be of the order of several microseconds, thus Δ is expected to be of the order of a few hundred microseconds. This is not deemed excessive if we realize that tolerance to Byzantine faults is being achieved.

6. Concluding Remarks

We have presented an architecture for triplicated processing of concurrent programs. The architecture has been devised for processors with point to point communication links. Assuming four (six) links per processor, a pipeline (grid) processor interconnection scheme has been proposed. We have developed a model for replicated processing and shown how replication transparency can be achieved. The underlying communications layer is required to support clock synchronization and atomic broadcasts between processor triads and provide a fairly conventional inter-process message passing facilities. A replication layer can then be built on top to perform ordering and voting of all inter-process messages between computational processes. As can be seen, exploiting redundancy in a distributed system to mask the effects of arbitrarily failing processors is a hard task! We are embarking on a project to build a Transputer based system with the characteristics discussed here. At the same time, we are investigating design issues for supporting time critical computations by enabling a process to deterministically select any message from its VMQ rather than just the one at the head . The prototype system will enable us to evaluate the cost of replication.

7. Acknowledgments

The work reported here has been supported in part by grants from the UK Science and Engineering Research Council.

References

[1] F.B. Schneider, "Abstractions for fault tolerance in distributed systems", Proc. of IFIP 86 Congress, 1986, pp. 727-733, North Holland.

[2] S.K. Shrivastava, "Replicated distributed processing", Lecture notes in Computer Science, Vol. 248, 1987, pp. 325-337, Springer Verlag.

[3] J.H. Wensley et al., "SIFT: Design and analysis of a fault-tolerant computer for aircraft control", Proc. of IEEE, 66, 10, October 1978, pp. 1240-1255.

[4] R.E. Harper, J.H. Lala and J.J. Deyst, "Fault tolerant processor architecture overview", Digest of papers, Fault Tol. Comp. Symp-18, Tokyo, June 1988, pp. 252-257.

[5] D. Powell et al., "The Delta-4 approach to dependability in open distributed computing systems", Digest of papers, Fault Tol. Comp. Symp-18, Tokyo, June 1988, pp. 246-251.

[6] K.P. Berman, "Replication and fault tolerance in the ISIS system", Proc. of 10th ACM Symp. on Operating System Principles, Washington, December 1985, pp. 79-86.

[7] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals problem", ACM Trans. on Prog. Lang. and Systems, 4(2), July 1982, pp. 382-401.

[8] Inmos Ltd, "Transputer Reference Manual", Prentice Hall, 1988.

[9] N. Theuretzbacher, "VOTRICS: Voting triple modular computing system", Digest of papers, Fault Tol. Comp. Symp-16, Vienna, July 1986, pp. 144-150.

[10] Inmos Ltd, "OCCAM 2 Reference Manual", Prentice Hall, 1988.

[11] A.D. Birrell and B.J. Nelson, "Implementing remote procedure calls", ACM Trans. on Computer Systems, 2(1), February 1984, pp. 39-59.

[12] R. Rivest, A. Shamir and L. Adleman, "A method of obtaining digital signatures and public-key cryptosystems", Comm. ACM, February 1978, pp. 120-126.

[13] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic broadcast: from simple message diffusion to Byzantine Agreement", Digest of papers, Fault Tol. Comp. Symp-15, Ann Arbor, June 1985, pp. 200-206.

[14] O. Babaoglu and R. Drummond, "(Almost) no cost clock synchronization (preliminary version)", Digest of papers, Fault Tol. Comp. Symp-17, Michigan, June 1987, pp. 42-47.

[15] L. Mancini, "Modular redundancy in a message passing system", IEEE Trans. on Soft. Eng., January 1986, pp. 79-86.

[16] K. Echtle, "Fault masking and sequencing agreement by a voting protocol with low message number", Proc. of 6th Symp. on reliability in dist. soft. and database systems, Williamsburg, March 1987, pp. 149-160.

[17] N. Natarajan and J. Tang, "Synchronization of redundant computation in a distributed system", Proc. of 6th Symp. on reliability in dist. software and database systems, Williamsburg, March 1987, pp. 139-148.

[18] L. Mancini and S.K. Shrivastava, "Failure detection in replicated systems", Computing Laboratory, University of Newcastle Upon Tyne, Tech. report no. 238, June 1987.

[19] L. Mancini and S.K. Shrivastava, "Exception handling in replicated systems with voting", Digest of papers, Fault Tol. Comp. Symp-16, Vienna, July 1986, pp. 384-389.

[20] J.Y Halpern, B. Simons, R. Strong and D. Dolev, "Fault tolerant clock synchronization", Proc. of 3rd Symp. on Principles of Dist. Computing, Vancouver, August 1984, pp. 89-102.