Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.C. Mulder

On the Amoeba protocol

Computer Science/Department of Software Technology Report CS-R8827 July

69C22, 69D13. 69D21, 69D22, 69F43

On the *Amoeba* Protocol

J.C. Mulder
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The *Amoeba* Distributed Operating system [Mul] supports the transaction as its communication primitive. The protocol that the *Amoeba* system uses to carry out sequences of transactions reliably and efficiently is analysed in terms of Process Algebra. The design goals are formulated as Process Algebra equations and it is established that one of them is not met. This can be repaired by adding an extra transition. Subsequently it is verified that the revised version meets its specifications.

1985 Mathematical Subject Classification: 68Q10, 68Q60, 68M10.
1980 Mathematical Subject Classification: 68B10, 68D25.
1982 CR Categories: C.2.2, D.1.3, D.2.1, D.2.2, F.4.3.
Note: Partial support received from the European Communities under ESPRIT project 432, An Integrated Formal Approach to Industrial Software Development (METEOR).

INTRODUCTION.

It has been observed that formal verification methods for mathematical proofs, computer programs, communication protocols and the like are usually illustrated by "toy" examples and that such proofs tend to be discouragingly long. In order to demonstrate that it is feasible to verify a "real-life" communication protocol by means of Process Algebra, we picked one from the literature.

In his Ph.D. thesis [Mul], Mullender investigates issues he considered while developing the *Amoeba* Distributed Operating System. In section 3.2.4 of [Mul] a transaction protocol is described to which we will refer as the *Amoeba* Protocol. In the preceding sections the design goals are described that this protocol is supposed to satisfy. He does not give a formal verification that his protocol meets this criteria. In fact, it turns out that one of them is not met. Note that this only applies to the simplified version of the protocol that appears in [Mul], the actual implementation uses a much more complicated version in which this mistake is not found.

Section 1 of this paper gives the minimum background information necessary for understanding the rest of the paper.
In section 2 the design goals are formulated in English and in terms of Process Algebra.
Section 3 describes the protocol and explains what is wrong.
In section 4 the (obvious) correction is given and it is verified that the resulting protocol meets the requirements.

The reader is supposed to be acquainted with Process Algebra. For an introduction we refer to e.g. [BK1,2,3].

ACKNOWLEDGEMENT. The author wishes to thank Frits Vaandrager for his careful proofreading and several improvements he suggested.

Report CS-R8827
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. Background information on *Amoeba* and Process Algebra.

1.0. It should be stressed that this section is not intended to give an accurate picture of the *Amoeba* System. We will only sketch the environment in which the *Amoeba* protocol operates. For a more detailed introduction the reader is referred to [Mul]. The reader is supposed to be acquainted with Process Algebra (see e.g. [BK1,2,3]), but for completeness an enumeration of the axioms used in this paper appears in section 1.14.

1.1. The context in which *Amoeba* operates is essentially a local area network connecting several machines with (possibly) different capabilities. E.g. some network nodes may have (or be) printers, huge disks, fast floating point hardware, etc. Needless to say, when a users posts a request, the system may decide to carry it out on another network node.

1.2. The centralised approach to such a configuration would use a request dispatcher residing on a fixed node in the center of the network. All requests would be mailed to the dispatcher, who would forward it to the machine that was most suitable for carrying it out. Of course this dispatcher must have up-to-date knowledge of work load, availability of services, etc.
This method is probably optimal in a star-shaped network, i.e. one in which one central machine is connected to all others and the others are connected to this central node only. In such a configuration all messages have to travel via the center node anyway.

However, in a more general network, the overhead of diverting each and every request via the center and keeping the dispatchers picture of the system up-to-date can probably better be avoided. Moreover, the central node might crash and it would be nice if the rest of the system would continue operating without it.

1.3. *Amoeba* uses a more distributed approach: each network node does its own dispatching. The *Amoeba* system does not try to maintain at every node a complete overview of what services are available on what nodes. If a user posts a request, the local *Amoeba* kernel may have to broadcast the question "which machines can carry out requests of type X?". Several machines may answer "I can" and then one of them is chosen; perhaps the first one to respond.

1.4. The *Amoeba* kernel does not carry out requests itself; it merely forwards them to a suitable server process, that may or may not live on the the machine. To the user the difference is immaterial, he is just posting requests and getting replies. In fact, a "user" may very well be a server process handing out a subtask of the request he is resolving.

1.5. In some network protocols, e.g. in the ISO model, the basic service is the virtual stream, carrying unlimited amounts of data from A to B. This may be very efficient when large amounts of data are to be transferred, but the designers of *Amoeba* felt that this would be a rare event in an *Amoeba* system. If, for instance, a user wants to query a large database, the database will not be transferred to the user, rather the query will be transferred to the database, thereby saving huge amounts of data transfer.

If it turns out that the information to be transferred in an average request fits in a single packet, then establishing and maintaining a virtual stream is not optimal.

1.6. The other end of the spectrum is a model in which the basic communication service is passing a single message. If the is to happen reliably, then for each message sent, an acknowledgement message must be bent back.

One might even be misled to think that this acknowledgement should also be acknowledged, and so on indefinitely. Fortunately, this is not the case: if the acknowledgement message does not arrive, then the sender of the original message will have to retransmit it. So if the receiver is able to recognise this retransmitted message as one it has received before, then it can simple reacknowledge it, and forget about it.

1.7. Nevertheless, *Amoeba* does not support the message passing primitive. The designers expect that most requests will lead to some sort of reply from the server, at the very least an indication of whether the request could be carried out. Obviously, a reply implicitly acknowledges receipt of the request. In fact, if the server has established his reply before the user feels like retransmitting his request, the original acknowledgement becomes superfluous.

To exploit the above possibility, *Amoeba* supports the *transaction* as its primitive communication service. This means that the process receiving the message (called the *server*) is obliged to send some sort of reply back to the sender (hereafter called the *client*). The client, on the other hand, is not obliged to return a follow-up query, so communication might stop after two messages.

1.8. The ISO communication standard prescribes some complicated seven-layer model. The *Amoeba* designers think that such a complicated system cannot possibly operate quickly, so they invented their own, three-layered model.

- The lowest layer is the Physical Layer. It consists of physical interconnections. We will not explore it any further.
- The middle layer is the Port Layer. The Port Layer transfers so-called "datagrams" of up to 32K to specified ports. A datagram is guaranteed to arrive *at most* once; it is left to the next higher layer to resubmit the datagram if necessary.
- The upper layer is the Transaction Layer. This is the layer we will investigate. It implements the transaction service, using the Port Layer's datagram service. If a datagram does not arrive the Transaction Layer will have to resubmit it. To detect such mishaps the Transaction layer employs the usual devices: timers and acknowledgement datagrams.

1.9. The Transaction Layer software on each network node has three interfaces: on the lower side there is an interface to the Port Layer and on the upper side there is an interface for clients and one for server processes.

When a client files a request, he indicates the type of service required by mentioning an associated port number. The Transaction Layer then uses the Port Layer for locating a server process offering this sort of service. If more then one server process offers this service, it is up to the Port Layer to pick a suitable one. Once this choice has been made, only four processes are relevant to the transaction from the Transaction Layer's point of view: the client, the server, the Transaction Layer software at the client's node, and the Transaction Layer at the server's node. In the sequel we will denote these as *CL*, *SV*, *TLCN* and *TLSN*, respectively.

1.10. In order to simplify the picture, we will largely ignore the fact that the Port Layer has to choose an available server able to carry out the specified type of request and act as if only the four processes mentioned above are involved. Obviously, we will concentrate on *TLCN* and *TLSN*, who try to communicate on behalf of *CL* and *SV* respectively via an unreliable medium provided by the Port Layer.

1.11. There is one more aspect of *Amoeba* that we do take into account. Sometimes, one of the network nodes crashes, i.e. it stops whatever it is doing and does not respond to any attempts to communicate.

We will assume that after a while this mishap is noticed and the malfunctioning machine is restarted. When restarted, the machine does not remember what is was doing before the crash, so it won't do anything before new requests arrive.

Consequently, when the *TLCN* has successfully delivered a request, it cannot simply wait for a reply. The network node where the server is working on a reply might crash and the client would be waiting forever. Instead, the *TLCN* will regularly poll the *TLSN* to check whether it is still alive. If the *TLSN* does not seem to be responding, the *TLCN* will assume that the server's network node has crashed and reissue the request, hoping that the unfortunate node has been restarted, or that some other server of the same type exists in the network. It might also notify some trouble server.

1.12. Conversely, it is not really a problem for a server if its client has crashed. If this happens the *TLSN* will be unable to deliver the reply, but once that fact has been discovered, there is not really any problem, thought the *TLSN* might notify the trouble server, just for the record. The server may have done some processing in vain, but that is tolerable, as crashes are rare events. Anyway, if the client had crashed immediately after receiving and acknowledging the reply, the result would have been the same: request carried out, result not used.

1.13. One might be tempted to think that, from a theoretical point of view, crashing machines are just another innocent feature, but this is not the case. If one wants to communicate reliably via an unreliable medium, one must be prepared to retransmit a message any number of times. If, on the other hand, one takes into consideration the possibility that one's partner has crashed, one should give up after a predetermined finite number of attempts. These options are evidently incompatible.

This does not necessarily imply that the *Amoeba* system is unreliable. In case of trouble the *TLCN* can usually restart the whole transaction. For some types of service it might be inappropriate to redo the essential processing. For example, suppose an accounting service keeps track of the usage of some services. Whenever one of the monitored servers satisfies a request, it notifies the accounting server. When the latter has updated its bookkeeping, it returns an acknowledgement. If this acknowledgement fails to be delivered, the accounting *TLSN* will assume that the other party has crashed. If this assumption is false, the *TLCN*, after a while, resubmit an account request. This glitch should not cause the user to be charged doubly, so the accounting server should be able to deduce from its books that it has satisfied this request before and react accordingly.

We will assume that this sort of safety precaution has been made and thus we will let the *TLCN* restart the whole transaction whenever it can not be completed satisfactorily.

1.14. We will assume that the reader is acquainted with Process Algebra. For completeness, we give its signature in table 1 below and the axioms we use in table 2 (next page).

a	atomic action ($a \in A$)
$+$	alternative composition (sum)
$\cdot$	sequential composition (product)
$\|$	parallel composition (merge)
$\lfloor\!\lfloor$	left-merge
$\mid$	communication merge
∂_H	encapsulation ($H \subseteq A$)
τ_I	abstraction ($I \subseteq A$)
δ	deadlock/failure
τ	silent action
π_n	projection ($n \in nat$)

TABLE 1. The signature of ACP_τ.

In table 2 the variables a, b and c range over $A \cup \{\delta\}$, x, y and z over processes, and H and $I \subseteq A$.

$x+y=y+x$	A1	$x\tau=x$	T1
$x+(y+z)=(x+y)+z$	A2	$\tau x+x=\tau x$	T2
$x+x=x$	A3	$a(\tau x+y)=a(\tau x+y)+ax$	T3
$(x+y)z=xz+yz$	A4		
$(xy)z=x(yz)$	A5		
$x+\delta=x$	A6		
$\delta x=\delta$	A7		
$a \mid b=b \mid a$	C1		
$(a \mid b) \mid c=a \mid (b \mid c)$	C2		
$\delta \mid a=\delta$	C3		
$x \parallel y=x \mathbin{\lfloor\!\lfloor} y+y \mathbin{\lfloor\!\lfloor} x+x \mid y$	CM1		
$a \mathbin{\lfloor\!\lfloor} x=ax$	CM2	$\tau \mathbin{\lfloor\!\lfloor} x=\tau x$	TM1
$ax \mathbin{\lfloor\!\lfloor} y=a(x \parallel y)$	CM3	$\tau x \mathbin{\lfloor\!\lfloor} y=\tau(x \parallel y)$	TM2
$(x+y) \mathbin{\lfloor\!\lfloor} z=x \mathbin{\lfloor\!\lfloor} z+y \mathbin{\lfloor\!\lfloor} z$	CM4	$\tau \mid x=\delta$	TC1
$ax \mid b=(a \mid b)x$	CM5	$x \mid \tau=\delta$	TC2
$a \mid bx=(a \mid b)x$	CM6	$\tau x \mid y=x \mid y$	TC3
$ax \mid by=(a \mid b)(x \parallel y)$	CM7	$x \mid \tau y=x \mid y$	TC4
$(x+y) \mid z=x \mid z+y \mid z$	CM8		
$x \mid (y+z)=x \mid y+x \mid z$	CM9	$\partial_H(\tau)=\tau$	DT
		$\tau_I(\tau)=\tau$	TI1
$\partial_H(a)=a \text{ if } a \notin H$	D1	$\tau_I(a)=a \text{ if } a \notin I$	TI2
$\partial_H(a)=\delta \text{ if } a \in H$	D2	$\tau_I(a)=\tau \text{ if } a \in I$	TI3
$\partial_H(x+y)=\partial_H(x)+\partial_H(y)$	D3	$\tau_I(x+y)=\tau_I(x)+\tau_I(y)$	TI4
$\partial_H(xy)=\partial_H(x)\cdot\partial_H(y)$	D4	$\tau_I(xy)=\tau_I(x)\cdot\tau_I(y)$	TI5

TABLE 2. The axioms of ACP$_\tau$.

ACP$_\tau$ is a complete axiomatisation of the identities between *finite* processes in the intended model (process graphs modulo rooted τ-bisimulation). However, several desirable principles are not derivable from ACP$_\tau$, even though all of their finite instances are. Moreover, they are true in the intended model. We will consider these as additional axioms. They are:

- RDP+RSP, the Recursive Definition and Specification Principle guarantee that *guarded* recursive specifications have a unique solution:

Let E be a guarded specification, then:	
(RDP) $\exists X\colon X \models E(Y)$	(RSP) $\dfrac{X \models E(Y) \quad X' \models E(Y)}{X=X'}$

- CFAR, the Cluster Fair Abstraction Rule.
  Let E be a guarded cluster, i.e. a specification of the form $\bigwedge_i X_i=\Sigma a_{ij}\cdot X_j+Y_i$ such that (i) all $a_{ij} \in I$, (ii) for each pair (X_i, Y_j) there is a sequence $X_{i_0}, \ldots, X_{i_n}$ s.t. $X_{i_0}=X_i$, Y_j is a summand of X_{i_n}, for all $k<n$: $a_{i_k i_{k+1}}\cdot X_{k+1}$ is a summand of X_{i_k} and this $a_{i_k i_{k+1}} \neq \delta$ and (iii) the system is *guarded*, i.e. there is no cycle $X_{i_0}, \ldots, X_{i_n}$ s.t. $i_0=i_n$ and for all $i<n$: X_{i_k} has a summand $\tau\cdot X_{i_{k+1}}$.
  The Y_j are called *exits* of the cluster.

The CFAR rule guarantees that the process specified by E will eventually leave the cluster:

$$\boxed{\text{(CFAR)} \quad \frac{X \models E(Y)}{\tau_I(X) = \tau \cdot \tau_I(\sum_j Y_j)}}$$

- HA, the Handshaking Axiom says that all communications are binary:

$$\boxed{\text{(HA)} \quad x \mid y \mid z = \delta}$$

- SC, the Standard Concurrency axioms:

$(x \mathbin{\lfloor\!\lfloor} y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} (y \parallel z)$	SC1
$(x \mid ay) \mathbin{\lfloor\!\lfloor} z = x \mid (ay \mathbin{\lfloor\!\lfloor} z)$	SC2
$x \mid y = y \mid x$	SC3
$x \parallel y = y \parallel x$	SC4
$x \mid (y \mid z) = (x \mid y) \mid z$	SC5
$x \parallel (y \parallel z) = (x \parallel y) \parallel z$	SC6

- ET, the Expansion theorem (in [BK3], it is derived from ACP$_\tau$ + HA + SC):

$$\boxed{\text{(ET)} \quad \parallel X = \sum_{x \in X} x \mathbin{\lfloor\!\lfloor} (\parallel (X - \{x\})) + \sum_{\substack{x, y \in X \\ x \neq y}} (x \mid y) \mathbin{\lfloor\!\lfloor} (\parallel (X - \{x, y\}))}$$

Here $\parallel \{x_1, ..., x_n\}$ abbreviates $x_1 \parallel \cdots \parallel x_n$.

- π_n, the projection operator:

$\pi_n(a) = a$	PR1	$\pi_n(\tau) = \tau$	PRT1
$\pi_1(ax) = a$	PR2	$\pi_n(\tau x) = \tau \cdot \pi_n(x)$	PRT2
$\pi_{n+1}(ax) = a \cdot \pi_n(x)$	PR3		
$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	PR4		

- α, the alphabet function:

$\alpha(\delta) = \varnothing$	AB1
$\alpha(\tau) = \varnothing$	AB2
$\alpha(ax) = \{a\} \cup \alpha(x)$	AB3
$\alpha(\tau) = \alpha(x)$	AB4
$\alpha(x + y) = \alpha(x) \cup \alpha(y)$	AB5
$\alpha(x) = \bigcup_{n \in \mathbb{N}} \alpha(\pi_n(x))$	AB6
$\alpha(\tau_I(x)) = \alpha(x) - I$	AB7

- CA, the Conditional axioms[1]:

$\dfrac{\alpha(x)\mid(\alpha(y)\cap H)\subseteq H}{\partial_H(x\parallel y)=\partial_H(x\parallel\partial_H(y))}$	(CA1)	$\dfrac{\alpha(x)\mid(\alpha(y)\cap I)=\varnothing}{\tau_I(x\parallel y)=\tau_I(x\parallel\tau_I(y))}$	(CA2)
$\dfrac{\alpha(x)\cap H=\varnothing}{\partial_H(x)=x}$	(CA3)	$\dfrac{\alpha(x)\cap I=\varnothing}{\tau_I(x)=x}$	(CA4)
$\dfrac{H=H_1\cup H_2}{\partial_H(x)=\partial_{H_1}\circ\partial_{H_2}(x)}$	(CA5)	$\dfrac{I=I_1\cup I_2}{\tau_I(x)=\tau_{I_1}\circ\tau_{I_2}(x)}$	(CA6)
$\dfrac{H\cap I=\varnothing}{\tau_I\circ\partial_H(x)=\partial_H\circ\tau_I(x)}$	(CA7)		

2. The Requirements

2.0. In this section we will try to pin down the design goals that the *Amoeba* system is supposed to satisfy, both in English and in terms of Process Algebra.

2.1. The main problem is to distinguish the three possible reasons why a client does not get an answer from a server:
(i) the server is still busy computing;
(ii) the server is trying to transmit a response, but the communication channel is malfunctioning; or
(iii) the server has crashed.

As explained in section 1, we assume that if a server crashes, so does its interface. Consequently, case (i) can be distinguished from (iii) by periodically polling the interface. If it reacts, we will assume that the server is still alive. On the other hand, as long as we don't get a response we know that either the channel malfunctions or the server has crashed (or both). If we fail to get a response a number of times successively, we find it highly unlikely that this is due to a faulty communication channel, so we assume that the server has crashed and start afresh. After a while the server will be in its initial state again, either because it had indeed crashed and is being restarted, or because it found that is was unable to deliver a reply to our original request.

In the real *Amoeba* system, the number of successive failures it takes before the client system decides to give up is fixed. In our presentation, whenever a client process fails to receive a sign of life it decides non-deterministically whether it will give up or try again.

2.2. Perhaps surprisingly, the hardest notion to catch in Process Algebra is periodical polling. The point is that Process Algebra does not explicitly mention time. After an event has happened, the next one takes place and there is no mention of the intervening time. If at a certain stage the only possible next event is that the server comes up with a result, this will be the next step in the process term, no matter how long it takes. Algebraically, we cannot say anything about the time spent waiting, because it does not appear in our formalism.

The only way to describe in Process Algebra that the interfaces exchange acknowledgements while waiting for the real reply, is saying that if the server never yields a reply, then infinitely many reacknowledgements will be transferred.

1. This is a misnomer. They should have been called 'Alphabet rules'.

2.3. A first approximation to an algebraic formulation of the above is the following:
Let *ans* be the event that the server delivers an answer; let *ack* be the event that the client receives an acknowledgement message; let I be the set of all other events; then we require, at least, that:

$$\tau_I \partial_{\{ans\}}(Amoeba) = \tau \cdot ack^{\omega}$$

2.4. The main defect of the above formula is that it describes the situation that every reply takes forever. In particular the first request will never be answered, so there will never be a second request. This can easily be repaired. It so happens that requests and replies will be indexed by natural numbers. Consequently, we can use $\partial_{\{ans(n)\}}$ to express that the server thinks infinitely long about the n^{th} request. So we will require (taking I to be all actions except *ack*):

$$\forall n \in \mathbb{N} \; \tau_I \partial_{\{ans(n)\}}(Amoeba) = \tau \cdot ack^{\omega}$$

To be quite honest, we should mention that a finite number of the acknowledgements mentioned above may have been exchanged while the server was contemplating the first $n-1$ requests.

2.5. The server network node runs (at least) two processes: the actual server process and its interface to the network. If this network node crashes, then both processes die simultaneously. This is not too hard to model. We introduce an atomic action *crash* and add to each and every term of the specification a summand $+crash$. Or rather, $+crash \cdot Server$, to model the fact that the server is eventually restarted by a crash server.
A minor complication is the fact that the specification is presented in the form $Server = Interface \| ServerProper$. We could, of course, introduce a yet another operator $x \rightarrow y$ that adds a summand y to every state of process x. In fact, in [B] such an operator is proposed under the name *mode transfer operator*. A similar operator occurs in LOTOS [ISO], where it is denoted $x[>y$ and called *disable operator*. But we can also manage by using existing operators. We introduce an atomic action *crash'*, that communicates with itself: $crash' | crash' = crash$, and we (textually) add summands $+crash'$ to all states of the interface and server proper.

2.6. The usual fairness assumptions in Process Algebra imply that, if the server is given the chance to crash infinitely often, it eventually will. One might interpret this as an instance of Murphy's Law, or regard it as a defect of Process Algebra. In any case, in this paper we will not propose any alternative notion of fairness. We will limit ourselves to verifying that the protocol does not abuse crashes to escape from problematic situations. In other words, for some suitable set I of internal actions, we will require:

$$\tau_I(Amoeba) = \tau_I \partial_{\{crash\}}(Amoeba)$$

2.7. The client process crashes in much the same way the server does. A minor difference is that a client process is not restarted when it crashes. As a result, the entire system will get stuck as soon as the server tries to communicate to its client again. Here our toy system with only one client deviates from the real *Amoeba* system where there is more than one client and one naturally requires that if one client dies, the server goes on to serve other clients. Thus we are led to also considering a two-client version and requiring

$$\tau_{I_2} \partial_H(Client_1 \| Client_2 \| Server) = \tau_I \partial_H(Client_1 \| Server)$$

where I_2 contains at least all actions pertaining to $Client_2$.

2.8. By now we have exhausted the requirements associated with crashing processes. We proceed by describing the regular behaviour of the system.

First of all, if the client does not crash, then it submits requests and the server should answer these. The requests are indexed by natural numbers and so are their responses. This gives something like:

$$\tau_I \partial_H(Amoeba) = \tau \cdot \prod_{n \in \mathbb{N}} req(n) \cdot ans(n)$$

Here, and in the sequel, we use $\prod_{n \in \mathbb{N}} T_n$ as a notation for a solution for X_0 of the system of equations $\{X_n = T_n \cdot X_{n+1} \mid n \in \mathbb{N}\}$.

The attentive reader has noticed that the above equation disagrees with our description in 2.7 in case of a client crash. A better approximation is

$$\tau_I \partial_H(\partial_{\{crash\}}(Client) \| Server) = \tau \cdot \prod_{n \in \mathbb{N}} req(n) \cdot ans(n)$$

This one, however fails to take into account, that a server crash may cause the request to be repeated. In fact, the easiest way out is to assume that the server never generates an answer unless it received a corresponding request and only specify that in the long run it is going to send all answers:

$$\tau_{I \cup \{req(n) \mid n \in \mathbb{N}\}} \partial_H(\partial_{\{crash\}}(Client) \| Server) = \tau \cdot \prod_{n \in \mathbb{N}} ans(n)$$

2.9. An important aspect we have been ignoring so far, is the communications channel connecting the *Client* and *Server* processes. In the *Amoeba* system this channel is set up and run by the Port Layer software. In Process Algebra, this channel is modelled as a separate process.

This *Channel* Process is described most easily as the parallel composition of two one-way channels. Such a one-way channel would accept a datum at one end and then choose non-deterministically between three options:

- deliver the datum at the other end undisturbed
- deliver it corrupted (this is assumed to be detectable)
- do not deliver anything at all.

In Process Algebra, this is easily described:

$$OWC = (read(datum) \cdot (i \cdot deliver(datum) + i \cdot deliver(error) + i)) \cdot OWC$$

In this equation i is an internal step, used as a guard. In an earlier paper, [KM], we used different guards for different options, but we now feel that this only opens up such weird possibilities as cutting out the second option by applying a $\partial_{\{i_2\}}$-operator. In fact, we would prefer to use τ as a guard here, but that is impossible.

2.9.1. If the one-way channel specified above could systematically choose, say, the second alternative, it would not be usable. Therefore, we will adopt the usual fairness rule, which implies that this cannot happen: if the same datum is input to the channel often enough, it will eventually be delivered correctly.

2.10. Incidentally, the *Amoeba* system does not respond at all if a corrupted message arrives. For one thing, one cannot extract the sender's name from a corrupted message. So the receiver is described in Process Algebra by a system of equations of the form:

$$Rec_k = accept(error) \cdot Rec_k + \sum a_l Rec_l$$

where the a_l are atomic actions distinct from *accept(error)*.

Now, if such a receiver is connected to a simplified one-way channel that does not deliver errors, but only "forgets" data, the result is the same as before:

LEMMA. Suppose processes *OWC*, *OWC'* and *Rec* are defined by the following systems of equations:

$$OWC = read(datum) \cdot OWC_2$$
$$OWC_2 = i \cdot OWC_3 + i \cdot OWC_4 + i \cdot OWC$$
$$OWC_3 = deliver(datum) \cdot OWC$$
$$OWC_4 = deliver(error) \cdot OWC$$

$$OWC' = read(datum) \cdot OWC'_2$$
$$OWC'_2 = i \cdot OWC'_3 + i \cdot OWC'$$
$$OWC'_3 = deliver(datum) \cdot OWC'$$

$$Rec_k = accept(error) \cdot Rec_k + \sum a_l Rec_l$$

Let

$$deliver(error) \,|\, accept(error) = arrives(error)$$
$$H = \{deliver(error), accept(error)\}$$
$$I = \{arrives(error)\}$$

Then

$$\tau_I \partial_H(OWC \| Rec) = \tau_I \partial_H(OWC' \| Rec)$$

PROOF: In every state the receiver can perform an action *accept(error)*. Execution of this action never results in a state change. Hence the receiver can be described in Process Algebra by a system of equations of the form:

$$Rec = accept(error)^{\omega} \| Rec'$$
$$Rec'_k = \sum a_l Rec'_l$$

Using the CA rules we find:

$$\tau_I \partial_H(OWC \| Rec) = \tau_I \partial_H(OWC \| Rec' \| arrives(error)^{\omega}) =$$
$$= \tau_I \partial_H(\tau_I \partial_H(OWC \| arrives(error)^{\omega}) \| Rec) = \tau_I \partial_H(OWC' \| Rec) = OWC' \| Rec \qquad \blacksquare$$

2.11. The upshot of all this is that there is no point in mentioning the possibility of the channel producing an error-value. Consequently, we will leave it out in the sequel. Thus the one-way channel will be described as:

$$OWC = read(datum) \cdot (i \cdot deliver(datum) + i) \cdot OWC$$

3. PROCESS ALGEBRA EQUATIONS

3.0. In this section we will present both the design criteria and the actual system in the form of Process Algebra equations. As it happens, the system presented in 3.2 does not satisfy the criteria in 3.3. This is easily mended and we will do so in 3.4.

3.1 Preliminaries. As a preliminary to the equations in 3.2, this subsection presents the alphabet, the communication function, etc.

3.1.1. The architecture. The architecture is depicted schematically in figure 3.1:

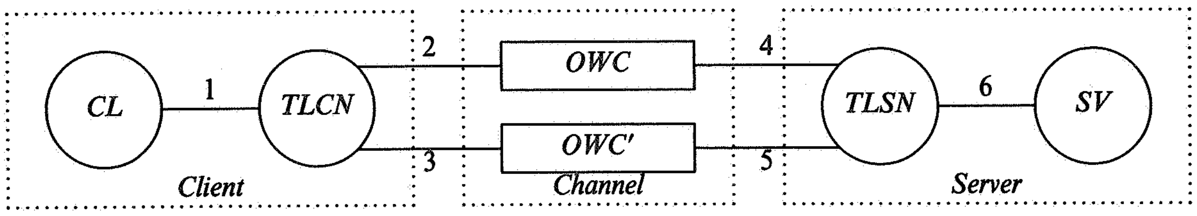


FIGURE 3.1

The *Amoeba* system contains a *Client* process and a *Server* process, connected by a communication channel. The *Client* process consists of a client proper and an interface. In 1.9 we have called these *CL* and *TLCN*, respectively. Likewise, the *Server* process consists of the server proper, *SV* and an interface, *TLSN*. Lastly there is the communication channel, that consists of two one-way channels.

These processes are connected by ports numbered as indicated in figure 3.1.

3.1.2. Data. Four types of messages are passed around: requests, answers, enquiries and acknowledgements. They will be denoted *req*, *ans*, *enq* and *ack*, respectively. For later reference, we collect them in a set $D = \{req, ans, enq, ack\}$. It does not make things clearer if we introduce actual contents for the requests and answers and consequently we will refrain from doing so.

In order to be able to describe the two-client version, we introduce the set $C = \{1,2\}$ of client numbers.

To make messages recognizable as pertaining to the same request, they are tagged with tags drawn from $T = C \times \mathbb{N}$, i.e. pairs consisting of the client's number and a sequence number.

We will need two auxiliary functions on T: if $t = (c,n)$, then $t^+ = (c, n+1)$ is the next tag from the same client and $t^- = (c, n-1)$ is the previous one (provided $n > 0$).

Messages passed on port 1 however, are not tagged, so the complete set of possible messages is $M = D \cup D \times T$.

3.1.3. Atomic actions. For $m \in M$ and $p \in \{1,...,6\}$ there are *read, send and communicate* actions:

$r(p,m)$: read message m at port p.
$s(p,m)$: send message m at port p.
$c(p,m)$: communicate message m at port p.

In fact $c(m,p)$ is a communication action: $c(p,m) = r(p,m) \mid s(p,m)$.
If the message m is in fact an element of $D \times T$, we will leave out some parentheses, and write e.g. $r(p,d,c,n)$ for $r(p, (d, (c,n)))$.
In section 2.5 we introduced the atoms *crash* and *crash′*, satisfying $crash = crash' \mid crash'$.
Finally we need an atomic action *to* denoting the timeout event and the communication channels contain an internal action *i*.

The entire alphabet is then:

$$A = \{r(p,m), s(p,m), c(p,m) \mid m \in M,\ 1 \leq p \leq 6\} \cup \{crash, crash', i, to\}$$

and the communication function is:

$$a \mid b = \begin{cases} c & \text{if } \exists m \in M, p \in \{1,\ldots,6\} \text{ s.t.} \{a,b\} = \{r(p,m), s(p,m)\} \wedge c = c(p,m) \\ crash & \text{if } a = b = crash' \\ \delta & \text{otherwise} \end{cases}$$

Some subsets of A will be referred to in the next subsection:

For $P \subseteq \{1,\ldots,6\}$: $H_p = \{r(p,m), s(p,m) \mid m \in M, p \in P\} \cup \{crash'\}$

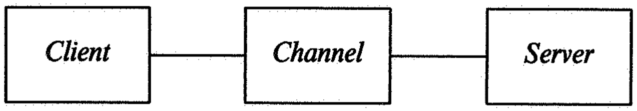
$$H = H_{\{1,\ldots,6\}}$$

$$I = A - \{c(3, ans, 1, n) \mid n \in \mathbb{N}\}$$

3.2 The specification

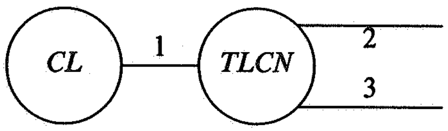
The *Amoeba* system consists of three component processes:

$$Amoeba = \partial_H(Client \| Channel \| Server)$$



We will describe these components in detail in the next three subsections.

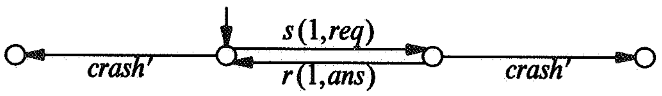
3.2.1. The Client process.



The *Client* process consists of the client proper and its interface:

$$Client = \partial_{H_{\{1\}}}(CL \| TLCN)$$

3.2.1.1. From our point of view, the client only generates requests and sometimes crashes:



$$CL = s(1,req) \cdot CL_2 + crash'$$

$$CL_2 = r(1,ans) \cdot CL + crash'$$

3.2.1.2. The client's interface, *TLCN*, accepts a request from the client process, gives it a sequence number and sends it to the server. If no answer arrives for some time, a timeout occurs and the request is sent again. If an acknowledgement arrives, the interface moves on to the next stage, where it periodically sends an enquiry message and expects another acknowledgement. If this acknowledgement fails to arrive, the *TLCN* non-deterministically chooses between sending another enquiry and believing that the server has crashed, in which case he starts afresh sending the request.

At any time during these stages, an answer to the request may arrive. The *TLCN* then delivers this answer, stripped of its tag, to the client and starts waiting for a further request. If the next request

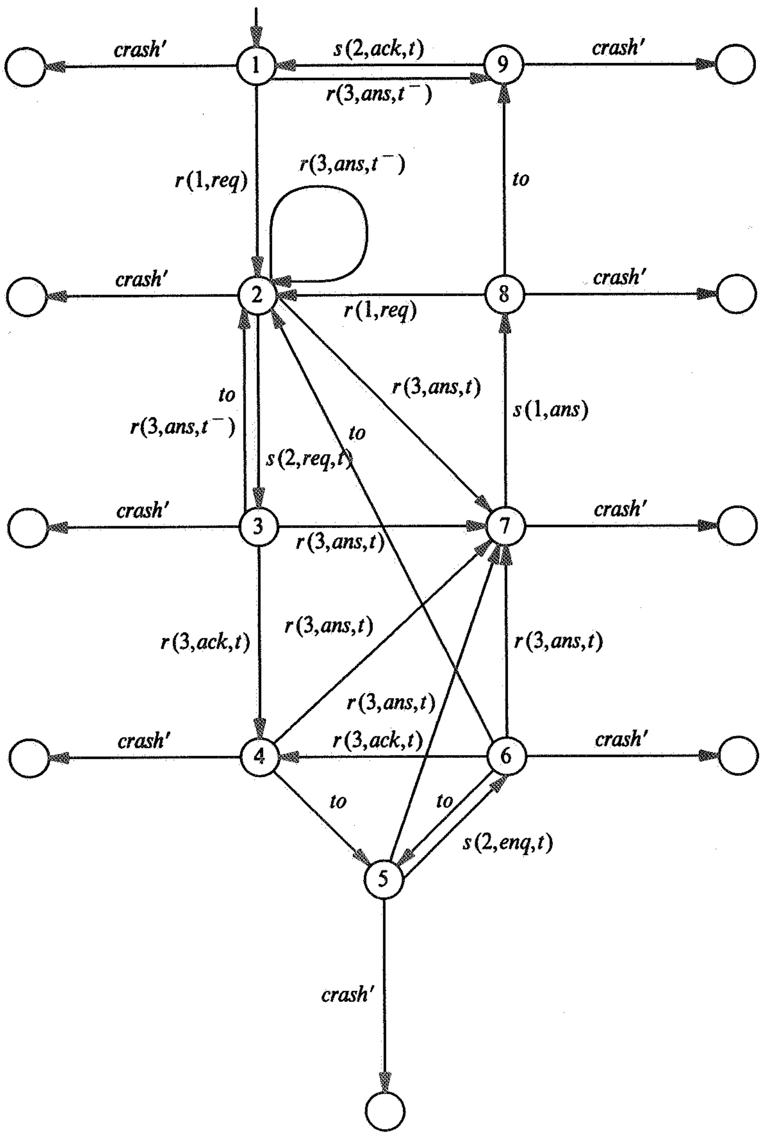


FIGURE 3.2. Process graph of the *TLCN*

comes quickly enough, the *TLCN* will enter the next cycle at the second stage, otherwise, it sends an acknowledgement and starts its next cycle at the beginning. In the first three states, it may happen that the answer to the previous question arrives again. The interface reacts by sending the current request, if it has one, and an acknowledgement otherwise.

The resulting process graph is shown in figure 3.1, except that the incrementing of the sequence number is not shown, in order to keep the picture finite.

This yields the following system of equations:

$$TLCN = TC_{1,(1,1)}$$

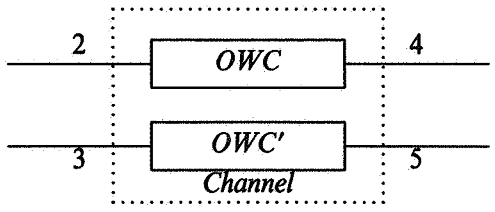
$$TC_{1,t} = r(1,req)\cdot TC_{2,t} + r(3,ans,t^{-})\cdot TC_{9,t^{-}} + crash'$$

$$TC_{2,t} = s(2,req,t)\cdot TC_{2,t} + r(3,ans,t^{-})\cdot TC_{3,t} + crash'$$

$$\begin{aligned}
TC_{3,t} &= to \cdot TC_{2,t} + r(3,ack,t) \cdot TC_{4,t} + r(3,ans,t) \cdot TC_{7,t} + r(3,ans,t^-) \cdot TC_{2,t} + crash' \\
TC_{4,t} &= to \cdot TC_{5,t} + r(3,ans,t) \cdot TC_{7,t} + crash' \\
TC_{5,t} &= s(3,enq,t) \cdot TC_{6,t} + r(3,ans,t) \cdot TC_{7,t} + crash' \\
TC_{6,t} &= to \cdot TC_{2,t} + to \cdot TC_{5,t} + r(3,ack,t) \cdot TC_{4,t} + r(3,ans,t) \cdot TC_{7,t} + crash' \\
TC_{7,t} &= s(1,ans) \cdot TC_{8,t} + crash' \\
TC_{8,t} &= r(1,req) \cdot TC_{2,t^+} + to \cdot TC_{9,t} + crash' \\
TC_{9,t} &= s(2,ack,t) \cdot TC_{1,t^+} + crash'
\end{aligned}$$

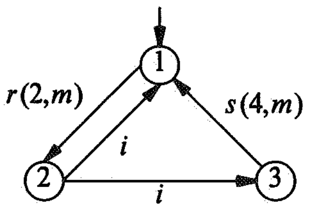
3.2.2. The Channel process. The channel consists two non-interacting one-way channels:

$$Channel = OWC \| OWC'$$



3.2.2.1. The one-way channel has been discussed at length in section 2.9. For reference we repeat:

$$\begin{aligned}
OWC &= \sum_{m \in M} r(2,m) \cdot OWC_{2,m} \\
OWC_{2,m} &= i \cdot OWC_{3,m} + i \cdot OWC \\
OWC_{3,m} &= s(4,m) \cdot OWC
\end{aligned}$$



3.2.2.2. The reverse channel is just a renaming of the first one:
Let $f: A \to A$ be the function defined by:

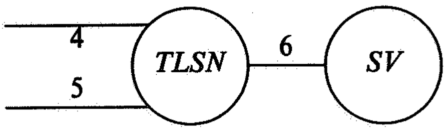
$$f(a) = \begin{cases} r(5,m) \text{ if } a = r(2,m),\ m \in M \\ s(3,m) \text{ if } a = s(4,m),\ m \in M \\ a \qquad \text{otherwise} \end{cases}$$

This f induces a renaming operator ρ_f and

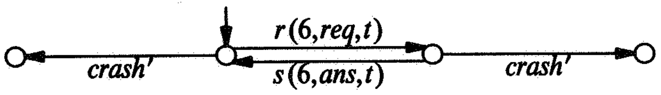
$$OWC' = \rho_f(OWC)$$

3.2.3. The Server process. The *Server* process consists of the server proper and its interface. If it ever stops, it is restarted:

$$Server = \partial_{H_{\{6\}}}(TLSN \| SV) \cdot Server$$



3.2.3.1. Like the client, the server is not studied in detail. It is just a sink of requests and a source of answers:



$$SV \;= \sum_{t \in T} r(6,req,t) \cdot SV_{2,t} + crash'$$

$$SV_{2,t} = s(6,ans,t) \cdot SV \;+ crash'$$

3.2.3.2. The server's interface is roughly analogous to its counterpart on the client's side. Initially, the interface awaits a request and relays it to the server. If the answer doesn't seem to come immediately, the interface acknowledges receipt of the request and awaits further events. If an enquiry message from an impatient client arrives, the *TLCN* waits some more and sends another acknowledgement. This gives rise to the following system of equations:

$$TLSN = TS_1$$

$$TS_1 \;= \sum_{t \in T} r(4,req,t) \cdot TS_{2,t} + \sum_{t \in T} r(4,enq,t) \cdot TS_1 + crash'$$

$$TS_{2,t} = s(6,req,t) \cdot TS_{3,t} + \sum_{m \in M} r(4,m) \cdot TS_{2,t} + crash'$$

$$TS_{3,t} = to \cdot TS_{4,t} + r(6,ans,t) \cdot TS_{6,t} + \sum_{m \in M} r(4,m) \cdot TS_{3,t} + crash'$$

$$TS_{4,t} = s(5,ack,t) \cdot TS_{5,t} + r(6,ans,t) \cdot TS_{6,t} + \sum_{m \in M} r(4,m) \cdot TS_{4,t} + crash'$$

$$TS_{5,t} = r(4,enq,t) \cdot TS_{3,t} + r(6,ans,t) \cdot TS_{6,t} + \sum_{\substack{m \in M \\ m \neq (enq,t)}} r(4,m) \cdot TS_{5,t} + crash' \qquad (*)$$

$$TS_{6,t} = s(5,ans,t) \cdot TS_{7,t} + \sum_{m \in M} r(4,m) \cdot TS_{6,t} + crash'$$

$$TS_{7,t} = r(4,req,t^+) \cdot TS_{2,t^+} + r(4,ack,t) \cdot TS_1 + to \cdot TS_{8,t} + \sum_{\substack{m \in M \\ m \neq (req,t^+) \\ m \neq (ack,t)}} r(4,m) \cdot TS_{7,t} + crash'$$

$$\begin{aligned} TS_{8,t} = {} & r(4,req,t^+) \cdot TS_{2,t^+} + r(4,ack,t) \cdot TS_1 + s(5,ans,t) \cdot TS_1 \\ & + s(5,ans,t) \cdot TS_{7,t} + \sum_{\substack{m \in M \\ m \neq (req,t^+) \\ m \neq (ack,t)}} r(4,m) \cdot TS_{8,t} + crash' \end{aligned}$$

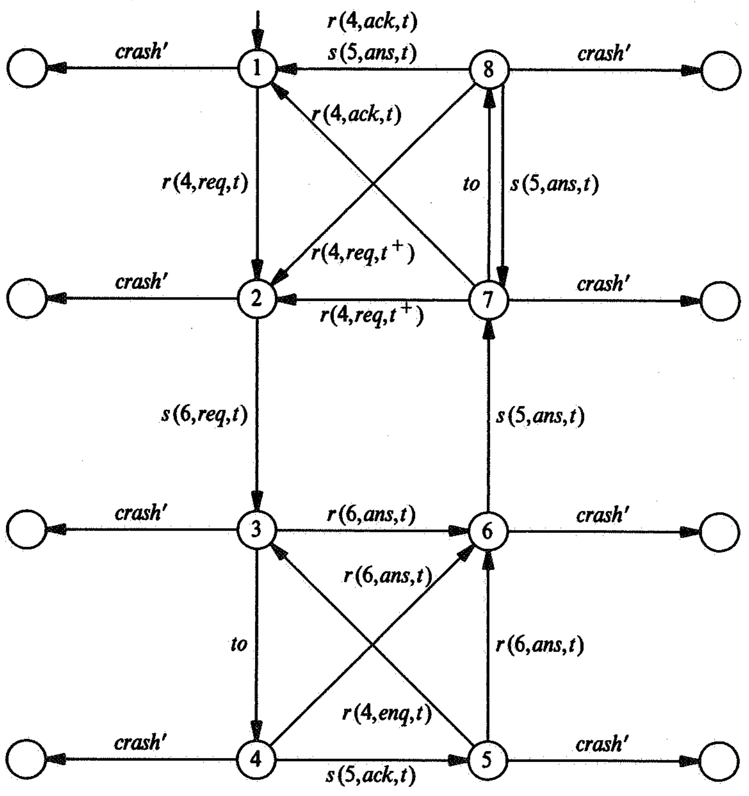


FIGURE 3.3. Process graph of the *TLSN*. Note that the loops $n \xrightarrow{r(4,m)} n$ are not shown.

Contrary to its counterpart on the client's node, the *TLCN* is always willing to accept and ignore the wrong input. The asymmetry arises because it is the client's rôle to get impatient if the computation takes a lot of time, and then generate a message that might be inappropriate by the time it arrives. These loops are not shown in figure 3.3. Also not shown is the incrementing of the sequence numbers.

Another minor difference is that the *TLSN* does not remember the request sequence number while the server is idle. The prime reason for doing so is to cope with the situation that a server has crashed and a new one is started. The new one should not insist on starting with request number 1, for that has already been served by its predecessor.

3.2.4. In the next subsection, we will study the subsystem consisting of the *Server* and the *Channel*, properly linked. For want of a better name, we will call this combination *System*:

$$System = \partial_{H_{\{4,5\}}}(Channel \| Server)$$

3.3. The requirements.

This section summarises the requirements from sections 2.3 through 2.8, with all i's dotted and some t's crossed. In particular, we will try to be clear on the position of the communication channel and the precise subsets of A occurring in various equations.

3.3.1. Client crashes apart, the client and server are exchanging requests and answers:

$$\tau_I \partial_{H_{(2,3)}}(\partial_{\{crash\}}(Client) \| System) = \tau \cdot \prod_{n \in \mathbb{N}} c(3, ans, 1, n)$$

where $I = A - \{c(3, ans, 1, n) \mid n \in \mathbb{N}\}$ as before.

3.3.2. While it is computing an answer, the *System* generates acknowledgement messages:

$$\tau_{I_n} \partial_{\{c(6, ans, 1, n)\}}(\partial_{\{crash\}}(Client) \| System) = \tau \cdot c(3, ack, 1, n)^{\omega}$$

where $I_n = A - \{c(3, ack, 1, n)\}$. Here we use the notation X^{ω} for $\prod_{n \in \mathbb{N}} X$. Recall that $c(6, ans, 1, n)$ is the event that the server establishes the n^{th} answer.

3.3.3. In a two-client system, even if one client crashes, the other will be served:

$$\tau_I \partial_H(Client \| \rho_g(Client) \| System) = \tau_I(Amoeba)$$

where $I = A - \{c(p, d, 1, n) \mid p \in \{2,3\}, d \in \{req, ans\}, n \in \mathbb{N}\}$,
and ρ_g is the renaming operator induced by $g: A \to A$ defined by:

$$g(a) = \begin{cases} r(p, d, 2, n) \text{ if } a = r(p, d, 1, n), & p \in \{2,3\}, d \in D,\ n \in \mathbb{N} \\ s(p, d, 2, n) \text{ if } a = r(p, d, 1, n), & p \in \{2,3\}, d \in D,\ n \in \mathbb{N} \\ a \qquad \text{otherwise} \end{cases}$$

Note that τ_I abstracts from (among others) all events pertaining to the second client.

3.3.4. Finally, the system does not abuse crashes to escape from illegal states. In other words, the equations above are also satisfied if the system is not allowed to crash:

$$\tau_I \partial_{\{crash\}}(Amoeba) = \tau \cdot \prod_{n \in \mathbb{N}} c(3, ans, 1, n)$$

$$\tau_{I_n} \partial_{\{crash, c(6, ans, 1, n)\}}(Amoeba) = \tau \cdot c(3, ack, 1, n)^{\omega}$$

$$\tau_I \partial_{\{crash\}} \partial_H(Client \| \rho_g(Client) \| System) = \tau_I \partial_{\{crash\}}(Amoeba)$$

3.4. When we tried to establish to establish algebraically that the Amoeba system satisfies the requirements in 3.3, we discovered that it does not.
To be specific, the system described in 3.2 does not satisfy the second requirement in 3.3.4. I.e., it may happen that the client and server's interfaces do not enter the phase where they exchange enquiries and acknowledgements while the server proper is establishing a reply. This situation ends when the replies comes, or if one of the parties crashes. If we block both these escapes we will observe livelock: the *TLCN* keeps repeating its request and the *TLSN* does not respond.

3.4.1. Trouble starts if the server's interface acknowledges receipt of a request, and this acknowledgement fails to arrive. When the *TLCN* times out, it assumes that the request was not delivered properly and repeats the request. The server's interface, however, is in a state where it does not accept requests on the ground that the server proper is busy. This interface is expecting an enquiry message. Thus, the parties are out of sync and will remain so until the server comes up with a reply, for that is the only event that both parties are willing to accept at their respective stages.

3.4.2. The shortest trace that leads to the problematic situation is:

$$c(1,req)\cdot c(2,req,1,1)\cdot i\cdot c(4,req,1,1)\cdot c(6,req,t)\cdot to\cdot s(5,ack,1,1)\cdot i$$

where the *to* is the one that takes the *TLSN* from state 3 to state 4 and the last *i* is the one in the $i\cdot\rho_f(OWC)$ summand of $\rho_f(OWC_{2,(ack,1,1)})$. This leads to a state:

$$S=\partial_{\{crash,c(6,ans,1,1)\}}\partial_H(\partial_{H_{\{1\}}}(CL_2\|TC_{3,1,1})\|OWC\|\rho_f(OWC)\|\partial_{H_{\{6\}}}(SV_{2,1}\|TS_{5,1,1}))$$

Inspecting the specifications, we see that the only possible transition in this state is a *to* that brings *TLCN* in state 2,1,1. Next, a $c(2,req,1,1)$ action brings *TLCN* back to state 3,1,1 and *OWC* to 2,$(2,req,1,1)$, from where it can choose between an *i* action or an *i* action followed by a $c(4,req,1,1)$ action. In both cases the global state reverts to *S*. Consequently, we are in a livelock situation.

3.4.3. The problem is easily mended. The point is that the *TLSN* ignores all requests while the server is busy. This policy is wrong: it should check whether the request is in fact a retransmission of the request the server is currently serving. If so, this indicates that the original acknowledgement message was lost in the return channel. Hence it should be retransmitted. In other words, a term $+r(4,req,t)\cdot TS_{3,t}$ should be added to the equation (⋆) for state $TS_{4,t}$ in section 3.2.3.2.

4. The verification

4.0. In this section we will formally verify that the corrected Amoeba protocol satisfies all requirements mentioned in 3.3. If the reader has read verifications in Process Algebra before, he will probably not find anything new in this section.

4.1. For reference, we include the revised version of the *TLSN* specification:

$$TLSN=TS_1$$

$$TS_1=\sum_{t\in T}r(4,req,t)\cdot TS_{2,t}+crash'$$

$$TS_{2,t}=s(6,req,t)\cdot TS_{3,t}+\sum_{m\in M}r(4,m)\cdot TS_{2,t}+crash'$$

$$TS_{3,t}=to\cdot TS_{4,t}+r(6,ans,t)\cdot TS_{6,t}+\sum_{m\in M}r(4,m)\cdot TS_{2,t}+crash'$$

$$TS_{4,t}=s(5,ack,t)\cdot TS_{5,t}+r(6,ans,t)\cdot TS_{6,t}+\sum_{m\in M}r(4,m)\cdot TS_{2,t}+crash'$$

$$TS_{5,t}=(r(4,enq,t)+r(4,req,t))\cdot TS_{3,t}+r(6,ans,t)\cdot TS_{6,t}+\sum_{\substack{m\in M\\ m\neq(enq,t)}}r(4,m)\cdot TS_{2,t}+crash'$$

$$TS_{6,t}=s(5,ans,t)\cdot TS_{7,t}+\sum_{m\in M}r(4,m)\cdot TS_{6,t}+crash'$$

$$TS_{7,t}=r(4,req,t^+)\cdot TS_{2,t^+}+(r(4,ack,t)+s(5,ans,t))\cdot TS_1+to\cdot TS_{8,t}+\sum_{\substack{m\in M\\ m\neq(req,t^+)\\ m\neq(ack,t)}}r(4,m)\cdot TS_{7,t}+crash'$$

$$TS_{8,t}=r(4,req,t^+)\cdot TS_{2,t^+}+r(4,ack,t)\cdot TS_1+s(5,ans,t)\cdot TS_{7,t}+\sum_{\substack{m\in M\\ m\neq(req,t^+)\\ m\neq(ack,t)}}r(4,m)\cdot TS_{8,t}+crash'$$

4.2. We start by showing that the presence of the client process is redundant:
LEMMA. *Client* $= \rho_h(TLCN)$
where ρ_h is the renaming induced by $h: A \rightarrow A$ defined by:

$$h(a) = \begin{cases} c(1,req) & \text{if } a = r(1,req) \\ c(1,ans) & \text{if } a = s(1,ans) \\ crash & \text{if } a = crash' \\ a & \text{otherwise} \end{cases}$$

PROOF. By direct calculation, one shows that the vector:

$$(\partial_{H_{(1)}}(CL \| TC_{1,t}),\ \partial_{H_{(1)}}(CL_2 \| TC_{2,t}),\ \ldots,\ \partial_{H_{(1)}}(CL_2 \| TC_{7,t}),\ \partial_{H_{(1)}}(CL \| TC_{8,t}),\ \partial_{H_{(1)}}(CL \| TC_{9,t}))$$

satisfies the (renamed) equations 3.2.1.2. The result then follows by RSP. ■

4.3. LEMMA. *Server* $= \rho_{h'}(TLSN)$
where $\rho_{h'}$ is the renaming induced by $h': A \rightarrow A$ defined by:

$$h'(a) = \begin{cases} c(6,req,t) & \text{if } a = s(6,req,t),\ t \in T \\ c(6,ans,t) & \text{if } a = r(6,req,t),\ t \in T \\ crash & \text{if } a = crash' \\ a & \text{otherwise} \end{cases}$$

PROOF. Analogous to 4.2. ■

4.4. For the sake of clarity, we will first consider the requirements from 3.3.4, the ones without server crashes. In this subsection, we will tackle:

$$\tau_I \partial_{\{crash\}}(Amoeba) = \tau \cdot \prod_{n \in \mathbb{N}} c(3, ans, 1, n) \qquad (*)$$

4.4.1. NOTATION. Let us denote:

$$Amoeba_n = \tau_I \partial_{\{crash\} \cup H_{(2,3)}}(\rho_h(TC_{2,1,n}) \| \partial_{H_{(4,5)}}(Channel_{n-1} \| \sum_{k \in \{1,7,8\}} \rho_{h'}(TS_{k,1,n-1}))) \quad \text{for } n > 1$$

where $Channel_n = OWC \| \rho_f(OWC_n)$ and $OWC_n = \sum_{\substack{k \in \{1,2,3\} \\ d \in D}} OWC_{k,(d,1,n)}$, where we take $OWC_{1,m}$ to mean OWC as specified in 3.2.2.1. We will write $OWC_{1,\varnothing}$ if we want to emphasise that the channel is actually empty. For $n = 1$ reference would be made to messages pertaining to request number 0. Because this does not exists, we have to define $Amoeba_1$ separately:

$$Amoeba_1 = \tau_I(Amoeba)$$

4.4.2. LEMMA. $Amoeba_n = \tau \cdot c(3, ans, 1, n) \cdot Amoeba_{n+1}$

The proof naturally breaks in two halves. Denote:

$$Halfway_n = \tau_I \partial_{\{crash\} \cup H_{(2,3)}}(\sum_{2 \leq k \leq 6} \rho_h(TC_{k,n}) \| \partial_{H_{(4,5)}}(OWC_n \| \rho_f(OWC_{n-1}) \| \rho_{h'}(TS_{6,n})))$$

4.4.3. LEMMA. For $n>1$: $Amoeba_n = \tau \cdot Halfway_n$

PROOF. Notice that the set of states

$$\{\partial_{\{crash\}\cup H_{\{2,3\}}}(\rho_h(TC_{i,n})\|\partial_{H_{\{4,5\}}}(OWC_{j,m}\|\rho_f(OWC_{k,m'})\|\rho_{h'}(TS_{l,n'}))) \mid 2\leqslant i\leqslant 6,\ j\leqslant 3,\ k\leqslant 3,\ m=(d,1,n),$$
$$l=1\vee(2\leqslant l\leqslant 5\wedge n'=n)\vee(7\leqslant l\leqslant 8\wedge n'=n-1),$$
$$d=req\vee d=enq,\ m'=(ans,1,n-1)\vee m'=(ack,1,n))\}$$

forms a (huge) cluster. The exits of this cluster are the summands of $Halfway_n$. The result now follows by CFAR. ■

4.4.4. LEMMA. $Amoeba_1 = \tau \cdot Halfway_1$.

PROOF. This is just a watered-down version of the previous lemma, the difference being that in this case the reverse channel cannot contain a message pertaining to the previous cycle. So the possible states are the elements of

$$\{\partial_{\{crash\}\cup H_{\{2,3\}}}(\rho_h(TC_{i,1})\|\partial_{H_{\{4,5\}}}(OWC_{j,m}\|\rho_f(OWC_{1,\varnothing})\|TS_{l,1}))$$
$$\mid i\leqslant 6,\ j\leqslant 3,\ l\leqslant 5,\ n=(d,1,1),\ d\in\{req,enq\}\}$$

As before, this forms a cluster, and the result follows by CFAR. ■

4.4.5. LEMMA. $Halfway_n = \tau\cdot c(3,ans,1,n)\cdot Amoeba_{n+1}$

PROOF. To begin with, the set of states

$$\{\partial_{\{crash\}\cup H_{\{2,3\}}}(\rho_h(TC_{i,n})\|\partial_{H_{\{4,5\}}}(OWC_{j,m}\|\rho_f(OWC_{k,m'})\|\rho_{h'}(TS_{l,n'}))) \mid 2\leqslant i\leqslant 6,\ j\leqslant 3,\ k\leqslant 3,\ l\in\{7,8,1\},$$
$$m=(d,1,n),\ d=req\vee d=enq,\ m'=(d',1,n'),\ d'=ack\vee d'=ans,\ n'\in\{n-1,n\}\}$$

forms a huge cluster, all of whose exits are of the form $c(3,ans,1,n)\cdot X$, with X an element of the set S below. Hence, by CFAR:

$$Halfway_n = \tau\cdot\sum_{X\in S} c(3,ans,1,n)\cdot X$$

where

$$S=\{\partial_{\{crash\}\cup H_{\{2,3\}}}(\rho_h(TC_{i,n})\|\partial_{H_{\{4,5\}}}(OWC_{j,m}\|\rho_f(OWC_{k,m'})\|\rho_{h'}(TS_{l,n'}))) \mid i\in\{7,8,9,1\},\ j\leqslant 3,\ k\leqslant 3,$$
$$l\in\{7,8,1\},\ m=(d,1,n),\ d\in\{req,enq,ack\},\ m'=(ans,1,n)\}$$

Again S is a cluster, so by CFAR: $\forall X\in S: X=\tau\cdot Amoeba_{n+1}$.

Summing up, we have:

$$Halfway_n=\tau\cdot\sum_{X\in S}c(3,ans\,1,n)\cdot X=\tau\cdot c(3,ans,1,n)\cdot\tau\cdot Amoeba_{n+1}=\tau\cdot c(3,ans,1,n)\cdot Amoeba_{n+1} \qquad ■$$

Equation (*) in 4.4 follows from the three lemmas above by observing that $\tau_I(Amoeba)=Amoeba_1$ (by definition) and cancelling all τ's except the initial one.

4.5. In this section we will establish the second equation from 3.3.4:

LEMMA. $\tau_{I_n}\partial_{\{crash,c(6,ans,1,n)\}}(Amoeba)=\tau\cdot c(3,ack,1,n)^{\omega}$.

Proof. As always, this boils down to applying CFAR to a suitable set of states, in this case:

$$\{\partial_{\{crash, c(6,ans,1,n)\}\cup H}(\rho_h(TC_{i,n})\|OWC_{j,m}\|\rho_f(OWC_{k,m'})\|TS_{l,n_2}) \mid i\leqslant 9,\ j,k\leqslant 3,\ l\leqslant 8,\ m'=(d',1,d_4)$$
$$m=(d,1,n_3),\ n_1,n_2,n_3,n_4\leqslant 5,\ d,d'\in D,\ n_2=n_1 \vee n_2=n_1-1,\ n_2\leqslant n_3\leqslant n_1,$$
$$n_4=n_2 \vee n_4=n_2-1,\ n_1=n\rightarrow i\leqslant 6,\ n_2=n\rightarrow l\leqslant 5,\ n_4=4\rightarrow d'\neq ans\}$$

As before, we have to convince ourselves that from each state in this set there is a path to an exit. After all, the trouble in the original specification was that from some states this was no longer possible. In this version it is: consider any state σ. If either channel contains a message, then the receiving process is willing to accept that message. So is it possible that these messages are delivered. Now, if the nth request hasn't yet been sent, it is possible that all requests up to the $(n-1)$th are sent and replied to promptly. Next, it is possible that the client (re)sends a $(req,1,n)$ or an $(enq,1,n)$ message. This may arrive and the *TLSN* may timeout and reply with ack $(ack,1,n)$, which may also arrive. So we see that from each state within the cluster there is a possible sequence of events leading to a $c(3,ack,1,n)$ exchange.

4.6. The third equation from 3.3.4:

$$\tau_I\partial_{\{crash\}}\partial_H(Client\|\rho_g(Client)\|System) = \tau_I\partial_{\{crash\}}(Amoeba)$$

At first one is tempted to use the CA rules to show that the τ_I-operator abstracts out all actions pertaining to the second client. However, this does not, and should not, work, as the second client interferes with the system. In fact, the point of the whole exercise is to show that the second client cannot clog up the system forever. Once we have established that the first client has a chance to proceed after finitely many steps of his colleague, and consequently infinitely many such chances, CFAR guarantees us that it will eventually proceed.

So we are to convince ourselves that:

$$\{\tau_I\partial_{\{crash\}}\partial_H(TC_{i,n_1}\|\rho_g(TC_{i',n_2})\|\partial_{H_{(4,5)}}(OWC_{j,m}\|\rho_f(OWC_{k,m'})\|\rho_{h'}(TS_{l,n_3}))) \mid i\leqslant 9,\ n_1\leqslant n,\ n_1=n\rightarrow i\leqslant 6,$$
$$i'\leqslant 9,\ n_2\in\mathbb{N},\ j,k\leqslant 3,\ l\leqslant 8,\ m=(d,c,p),\ d=req \vee d=enq,$$
$$c=1\wedge p=n_1 \vee (c=2\wedge p=n_{2,}),\ m'=(d',c',p'),\ d'=ans \vee d'=ack,$$
$$(c'=1\wedge(p'=n_1 \vee p'=n_1-1)) \vee (c'=2\wedge(p'=n_2 \vee p'=n_2-1)),\ n_3\in\{n_1,n_1-1,n_2,n_2-1\}\}$$

indeed forms a cluster with exits of the form $c(3,ans,1,n)\cdot X$.

The hard part here is to convince oneself that from each point within this cluster there is a path to an exit. If we compare the specifications of the channels and the *TLxN*s we can see that in each of the states mentioned above it is possible that the channels deliver any messages they may contain; next, if the $\rho_{h'}(TLSN)$ is not in its initial state, it is possible that it completes the transaction it is dealing with, and if that was not the nth transaction for *Client* 1, then it is possible that the system goes on to carry out transactions with *Client* 1, until it has completed the nth. So we see that indeed from each point in the cluster there is a path to an exit.

4.7. Having satisfied ourselves that the system behaves as promised if it doesn't crash, we turn to cases where the server does crash and is restarted from scratch. If this happens while the old server was busy and the client knew this, i.e. it had received an acknowledgement but not yet an answer, the client will send a number of *enq* messages, to which the server doesn't respond[2]. After a while the client guesses what has happened, resends its request and reverts to state 2,*n*.

What we set out to verify in this section, is that if the client makes this cycle through states 2-6 and

2. In the data set as described in [Mul] there is a *nak* message which seems applicable here, but there is no mention of it ever being used. In the actual *Amoeba* system it is, of course, used in this situation.

the client possibly crashes, then the parallel composition of these four processes just runs around some gigantic cluster and never gets stuck in a dark corner thereof.

4.7.1. In order to establish equation 3.3.1:

$$\tau_I \partial_{H_{\{2,3\}}}(\partial_{crash}(Client) \| System) = \tau \cdot \prod_{n \in \mathbb{N}} c(3, ans, 1, n)$$

we denote:

$$Amoeba_n = \tau_I \partial_{H_{\{2,3\}}}(\partial_{\{crash\}}(\rho_h(TC_{2,n})) \| \partial_{H_{\{4,5\}}}(Channel_{n-1} \| \sum_{l \in \{1,7,8\}} \rho_{h'} TS_{k,n-1})))$$

where $Channel_n$ is the same as in 4.4.1. (note that $Amoeba_n$ is not the same).

4.7.2. LEMMA. $Amoeba_n = \tau \cdot c(3, ans, 1, n) \cdot Amoeba_{n+1}$

PROOF. The set of states

$$\begin{aligned}\{\partial_{H_{\{2,3\}}}(\partial_{\{crash\}}(\rho_h(TC_{i,n}) \| \partial_{H_{\{4,5\}}}(OWC_{j,m} \| \rho_f(OWC_{k,m'}) \| \rho_{h'}(TS_{l,n'}))) \mid 2 \leqslant i \leqslant 6,\ j \leqslant 3,\ k \leqslant 3,\ l \leqslant 8,\\ n' = n \vee (7 \leqslant l \leqslant 8 \wedge n' = n-1),\ m = (d, 1, n),\ d = req \vee d = enq,\\ m' = (ans, 1, n-1) \vee m' = (ack, 1, n) \vee m' = (ans, 1, n)\}\end{aligned}$$

forms a cluster whose exits are of the form $c(3, ans, 1, n) \cdot X$, with X in the set S below. In particular, if the server crashes and is restarted, it goes to state $1, n$, which is still in this cluster. Applying CFAR yields:

$$Amoeba_n = \tau \cdot \sum_{X \in S} c(3, ans, 1, n) \cdot X$$

where

$$\begin{aligned}S = \{\partial_{H_{\{2,3\}}}(\rho_h(\rho_h(TC_{i,n})) \| \partial_{H_{\{4,5\}}}(OWC_{j,m} \| \rho_f(OWC_{k,m'}) \| \rho_{h'}(TS_{l,n'}))) \mid i \in \{7,8,9,1\},\ j \leqslant 3,\ k \leqslant 3,\\ l \in \{7,8,1\},\ m = (d, 1, n),\ d \in \{req, enq, ack\},\ m' = (ans, 1, n)\}\end{aligned}$$

S is a cluster, too, and by CFAR $\forall X \in S : X = \tau \cdot Amoeba_{n+1}$. So we conclude:

$$Amoeba_n = \tau \cdot c(3, ans, 1, n) \cdot Amoeba_{n+1}$$

■

4.7.3. To complete the proof of 3.3.1, we have to show that $\tau_I(Amoeba) = \tau \cdot c(3, ans, 1, 1) \cdot Amoeba_2$. In this case the reverse channel cannot contain a message $(ans, 1, 0)$, so the cluster simplifies to:

$$\begin{aligned}\{\partial_{H_{\{2,3\}}}(\partial_{\{crash\}}(\rho_h(TC_{i,1}) \| \partial_{H_{\{4,5\}}}(OWC_{j,m} \| \rho_f(OWC_{k,m'}) \| \rho_{h'}(TS_{l,1}))) \mid 2 \leqslant i \leqslant 6,\ j \leqslant 3,\ k \leqslant 3,\ l \leqslant 8,\\ m = (d, 1, 1),\ d = req \vee d = enq,\ m' = (ack, 1, 1) \vee m' = (ans, 1, 1)\}\end{aligned}$$

The rest of the proof is entirely analogous to 4.7.2. ■

4.7.4. The proofs of 3.3.2 and 3.3.3 are entirely analogous to those in 4.5 and 4.6 and will therefore be omitted. The point is that even if the server crashes and begins afresh, the system does not leave the relevant cluster. The *crash* transition provides an extra path from certain states to the exit, but we have already established in 4.5 and 4.6 that such paths exist in the revised version of the protocol.

5. Conclusions.

We have demonstrated that it is possible to describe in Process Algebra equations what liveness and safety properties a communication protocol is meant to have. Theoretically it is also possible to derive that a specific implementation indeed possesses these properties but this usually boils down to apply the CFAR rule to large and intricate clusters and Process Algebra offers little means to handle these gracefully. One seems to need a criterion that guarantees that all states satisfying some assertion form a cluster and that CFAR may be applied to it. The state operator [BB] might provide such a means, but as of yet we do not see how to use it for this purpose.

References.

[B] J.A. BERGSTRA, *A Mode Transfer Operator in Process Algebra,* report P8808, Programming Research Group, University of Amsterdam, 1988.

[BB] J.C.M. BAETEN & J.A. BERGSTRA, *Global Renaming Operators in Concrete Process Algebra,* (revised version), Programming Research Group report P8709, University of Amsterdam, Amsterdam, 1987. To appear in Information & Computation.

[BK1] J.A. BERGSTRA & J.W. KLOP, *Process Algebra for Synchronous Communication,* Information & Control 60 (1/3) , pp 109-137, 1984.

[BK2] J.A. BERGSTRA & J.W. KLOP, *Process Algebra: Specification and Verification in Bisimulation Semantics.* in: *Mathematics and Computer Science II,* CWI monograph 4, (M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens, eds.), North-Holland, Amsterdam, pp 61-94, 1986.

[BK3] J.A. BERGSTRA & J.W. KLOP, *Algebra of Communicating Processes,* in: *Proceedings of the CWI Symposium Mathematics and Computer Science* (eds. J.W. de Bakker, M. Hazewinkel & J.K. Lenstra), North-Holland, Amsterdam, pp 89-138, 1986.

[ISO] ISO *Information processing systems – Open systems interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour* (ed. Ed Brinksma), ISO/TC97/SC21N DIS8807, 1987.

[KM] C.P.J. KOYMANS & J.C. MULDER, *A Modular Approach to Protocol Verification using Process Algebra,* RUU report LGPS 6, University of Utrecht, Utrecht, 1986. To appear in: *Applications of Process Algebra,* (ed. J.C.M. Baeten).

[Mul] S.J. MULLENDER, *Principles of Distributed Operating System Design,* Ph.D. thesis, Free University, Amsterdam, 1985.