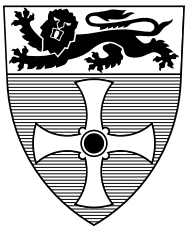
UNIVERSITY OF
NEWCASTLE



COMPUTING SCIENCE

Engineering Look-ahead in Distrib Conversations

Paul Ezhilchelvan and Alexander Romanovsky

TECHNICAL REPORT SERIES
No. CS-TR-659
February, 1999

Contact:Paul Ezhilchelvan and Alexander Romanovsky
[Paul.Ezhilchelvan@ / Alexander.Romanovsky @ncl.ac.uk
[http://www.cs.ncl.ac.uk/people/paul.ezhilchelvan/
http://www.cs.ncl.ac.uk/people/alexander.romanovsky/]

[A shortened version of this report appears in the Proceedings of International Symposium on Autonomous Decentralised Systems (ISADS Mar 21 -23, 1999.]

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

Engineering Look-ahead in Distributed Conversations

Paul Ezhilchelvan and Alexander Romanovsky

Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

This paper investigates the effects of relaxing the synchronisation embedded in "classical" conversation schemes. Look-ahead conversation scheme [KY89] enables the synchronisation mandated by conversations to be performed concurrently to other normal system activities, and thereby provides scope for enhancing system performance. In this paper, we take the view that permitting look-ahead must guarantee that executions with and without look-ahead be equivalent for identical inputs and run-time conditions. We identify and formulate the necessary condition for meeting this objective. We then present two schemes for realising this condition. The first scheme is based on piggybacking extra information onto ongoing messages, and the second one is a message passing protocol requiring each participant to send one message to every other participant of the conversation. These schemes do not require a conversation participant to know *a priori* all participants of the conversation, but only those it is designed to interact with, during the conversation.

Keywords: fault-tolerance, distributed systems, conversations, synchronised exit, look-ahead scheme.

1. Introduction

We consider a system of autonomous decentralised components, whose output, once produced, affects the environment either irreversibly or in a manner that is expensive to compensate. Such a system needs to be fault tolerant and must be effectively structured to manage the additional complexity introduced by fault-tolerance activities. Conversations [R75] provide a convenient structuring concept for introducing fault tolerance into systems. System activity is composed of units of recoverable component activities, called *conversations*. Two or more components engage in a conversation which either terminates acceptably or appears not to have been carried out at all. The latter outcome occurs when a software fault manifests during execution and its occurrence is minimised by having redundant software within the participating components. A simplified unmanned vehicle control system [YK92], an anti-missile defence C3 application system [KB97] and a series of production cell case studies [ZR98, XR98] are examples of systems designed using the conversation concept.

Structuring system activity into computational units that have specific, well-defined properties - be it by conversations (typically in process control) or by Transactions [G78] (typically in database and business applications) - requires components to synchronise their activities in order that some of those properties can be guaranteed, e.g., the atomicity property of transactions is achieved through 2-phase commit protocol. This inevitable synchronisation extracts a cost when components are autonomous and distributed: fast components need to wait for slow ones to catch-up. The *look-ahead* scheme [KY89] attempts to reduce this cost of synchronisation during conversation executions. In this paper, we take the view that permitting look-ahead should ensure that executions with and without look-ahead be equivalent for identical inputs and run-time conditions. With this objective in mind, we propose a way to implement the look-ahead scheme.

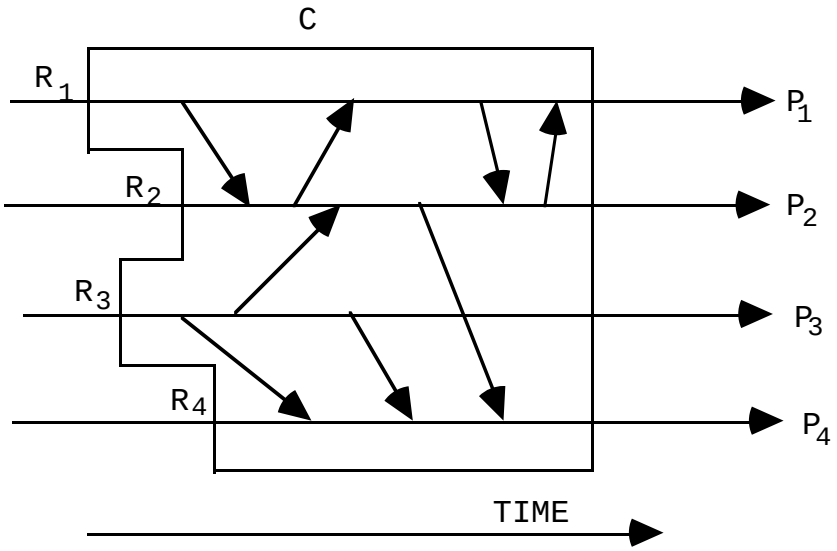


Figure 1: Basic conversation scheme.

The working of the basic conversation scheme is illustrated in Figure 1. Processes P_1, P_2, P_3 and P_4, known as *participants*, are involved in a conversation C. Each P_i, i=1..4, enters C after establishing recovery point R_i. Once inside C, it executes the prescribed software and exchange messages only with the participants that have made entry into C. Flow of messages are indicated in Figure 1 by slanted arrows. A participant P_i's (horizontal) time line has a vertical line at the beginning and at the end of C; they respectively indicate the time when P_i establishes and discards R_i. Participants can enter C at different times (i.e. asynchronously) but they leave C synchronously, only after it is known that they all have passed their local acceptance tests (LATs). Even if one participant fails the LAT, all participants *roll back* by restoring their respective R_i, and try the next alternate (if any) of their redundant software. Thus, the rules of conversation ensure that the states of a process within a conversation is not 'visible' to any process outside the conversation, thus avoiding the undesired *domino effect* [R75]; also that the process activities within a conversation are 'atomic' against process failures: they either complete correctly with respect to LATs or appear not to have been

carried out at all. A conversation is said to be *validated* when all participants pass their LATs. Once a participant learns of the validation, it discards its recovery point.

A conversation may be *nested* within another conversation. A subset of the participants of a conversation (C in Figure 2) enter another conversation (C_I in Figure 2) before they check the LATs for the first conversation. This form of conversation nesting requires that the second, nested conversation be validated before the LATs for the first one (C) is checked. In this paper, we call the conversation that contains a nested conversation an *enveloping* conversation. A nested conversation can envelop another nested conversation, and so on. In theory, there is no limit on the degree of nesting.

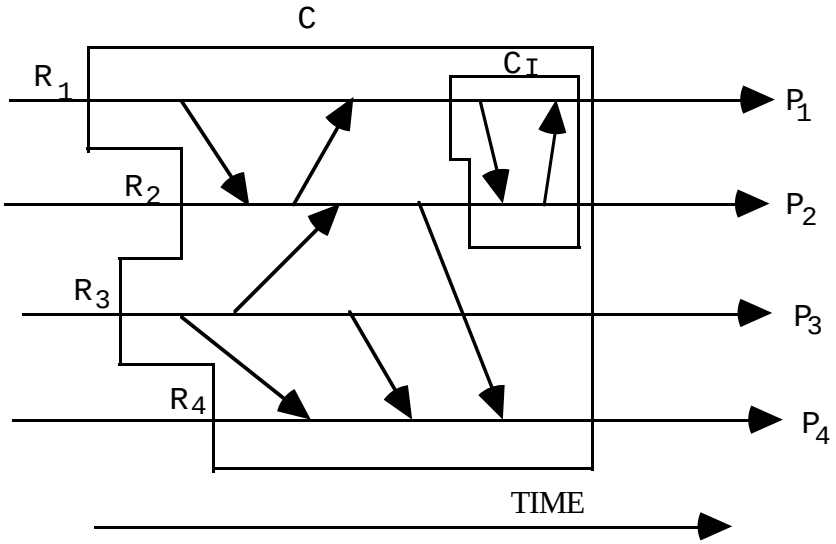


Figure 2: C_I is nested within C.

In this paper, we concentrate on conversations of distributed participants. [KY89] systematically analyses the cost of implementing distributed conversations and identifies that the cost of synchronising participants can substantially increase the time overheads. The intuitive reasoning for this observation is as follows: message passing in a distributed system can be slow and the processing speeds of distributed participants can differ widely. Synchronisation requires that fast participants wait for the slow ones for conversation to be validated. Thus, the slowest participant ends up determining the speed of conversation execution. Assuming that there is no front-end synchronisation (as in the basic scheme), [KY89] proposes a novel, look-ahead scheme to circumvent the rear-end synchronisation where possible.

The focus of our paper is to analyse the effect of incorporating look-ahead into distributed conversations, and to identify the necessary condition to guarantee that an execution with look-ahead produces the same outcome as the execution without lookahead, when both executions are carried out in identical run time conditions. This analysis considers that processes may be programmed with non-deterministic message selection: a participant waits on timeout to receive a message from one or many sources; upon receiving the first message or timeout, it resumes the activity. Non-deterministic message selection is not uncommon in ADS and is a useful way to minimise process

waiting and hence to improve system performance. We present schemes for meeting the necessary condition, in a context where a participant need not know *a priori* the ids of all other participants; it only has to know those participants with which it is designed to interact with, during the conversation.

Our investigation is performed in the most generic of the models proposed by [YK92]: at the end of the conversation, each participant executes its LAT and disseminates the result of that execution to every other participant it knows of. Based on the information received, each participant executes a validation protocol to identically decide whether the conversation is validated or needs to be rolled back. This decentralised approach to detecting validation is obviously message intensive; the other, centralised and semi-centralised approaches of [YK92] incur less message overhead and are, from a conceptual point of view, variations of decentralised approach. For that reason, we consider only the decentralised approach in our basic model used for our investigation and the solutions we propose here remain equally valid for other approaches as well.

The paper is organised as follows. Section 2 discusses the look-ahead scheme in detail. Section 3 introduces system and conversation models, and formulates the characteristics of properly formed conversations. Section 4 discusses an example which shows that look-ahead can give rise to partial orderings of the system events which can never happen in systems without look-ahead, and can result in incorrect system behaviour - a behaviour that would never be exhibited if look-ahead were disallowed**.** The general rule necessary to solve this problem is formulated in Section 5; this Section discusses two schemes to implement the rule. Conclusions are presented in Section 6.

2. Principles of Look-ahead

The basic ideas behind the look-ahead scheme are as follows. The look-ahead scheme permits a participant to leave the conversation soon after it passes its LAT for that conversation, even though it cannot know at that time whether the conversation is going to be validated or rolled back. It must however not discard the recovery point so that it does not lose the ability to roll back in case one of the other participants fails its LAT. After leaving the conversation in this safe manner, it can enter a new conversation and the participants of the second conversation may include some of the old participants which have also similarly left the first conversation. Thus, the look-ahead scheme enables fast participants to proceed ahead without waiting for the slow ones and thus provide scope for improved system performance.

When a participant P_i enters a conversation C' before knowing the validation of C for which it has passed the LAT, we will say that P_i *looks ahead* (from C) into C'. In Figure 3, P_3 looks ahead into C_2 from C_1. A broken vertical line on a participant's time line indicates the time the participant passes only the LAT for a conversation; the first subscript of R indicates the participant id and second the conversation name. Suppose that P_3 and P_4 pass their respective LATs for C_2 before C_1 can be validated. P_4 disseminates this fact to P_3 in an attempt to validate C_2. If P_3 also does the same and if P_4 knows that P_3 and itself are the only participants of C_2, then P_4 will rightly conclude that C_2 is validated and discard R_{42}. This may cause problems if, say, P_1

fails the LAT for C_1 and P_3 is required to roll back C_1; P_3 restoring R_{31} requires P_4 to restore R_{42} which is now impossible. (Note that if P_4 is acting as the head participant in a centralised scheme, it will discard R_{42} after receiving the information that P_3 has passed the LAT for C_2; so, this situation is not unique to the decentralised approach.) To prevent such premature validation of looked-ahead conversations, a participant is prevented from disseminating the information of it having passed the LAT for a looked-ahead conversation until it knows that all conversations for which it has earlier passed the LATs have been validated.

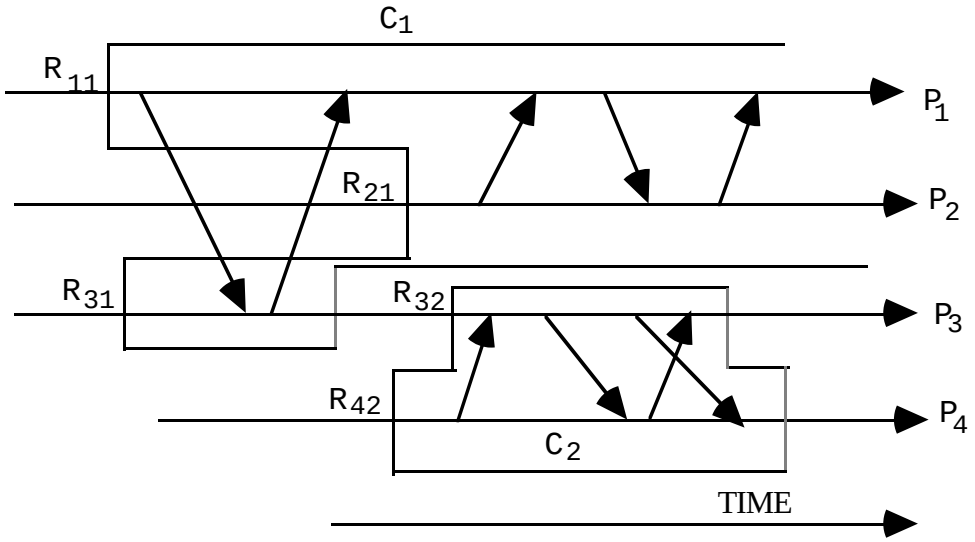


Figure 3: P_3 looks ahead into C_2 from C_1.

The main disadvantage of look-ahead is that in the worst case, a fast process can be through several yet-to-be-validated conversations, all of which will have to be discarded if a slow participant fails its LAT. Moreover processes which are not participants of the failed conversation (such as P_4 in Figure 3) will have to take part in the rollback as well. However, the analysis in [KY89] shows that performance gain is possible if we restrict the number of conversation executions a process can look ahead; with optimal number of permitted look-ahead, the performance is improved when failures (and hence rollbacks) are rare. A practical conclusion that can be made from this analysis is that a system should be designed with look-ahead option, and be commissioned initially without look-ahead; as the software bugs that get exposed are fixed, a stage will be reached when it is beneficial to employ look-ahead for performance gains. This switch-over can be done safely, however, only if it is guaranteed that the executions with and without look-ahead produce identical outcomes when the run time conditions are identical.

3. System Description and Assumptions

A conversation, named *c*, involves a set of participants, Part(*c*). We assume that each participant executes on a distinct, reliable node in a distributed system. (Assuming a

reliable execution platform enables us to concentrate only on failures caused by bugs in application software[1].) Each participant has an ordered set of redundant programs to execute in *c* and an LAT to verify its execution. A participant P_i is said to *communicate directly* with another P_j in *c*, if it has been designed to send messages to or receive messages from P_j during any execution of *c*. Let $Comm_i(c)$ denote the set of all participants with which P_i communicates directly in any execution of *c*. $Comm_i(c) \subseteq Part(c)$ and is non-empty for any P_i; also, if $P_j \in Comm_i(c)$ then $P_i \in Comm_j(c)$: $P_j \in Comm_i(c) \Leftrightarrow P_i \in Comm_j(c)$.

At the start of the system, P_i is assumed to know $Comm_i(c)$, but not necessarily $Part(c)$. Not enforcing P_i to know $Part(c)$ simplifies the task of modifying *c* to include a new participant: the codes of those (distributed) participants that do not communicate directly with the new participant, need not be inspected as they require no modification.

P_i is said to *communicate transitively* with P_j in *c*, if for some $n \geq 1$:

$P_j \notin Comm_i(c)$ and

$\exists$ k1, k2, ..., kn: $P_j \in Comm_{k1}(c) \wedge P_{k1} \in Comm_{k2}(c) \wedge ... \wedge P_{kn} \in Comm_i(c)$.

Since a conversation is a unit of activity in which *all* participants cooperate with each other, we assume the following *closure property*: every P_i communicates either directly or transitively with every other P_j. In other words, there does not exist a set S, $S \subset Part(c)$, such that for all P_i in S: $\cup Comm_i(c) = S$. If such a set S exists then this would mean that processes in S never have to cooperate with those in $Part(c) - S$. This is a sign of bad structuring: *c* is composed of non-interacting subsets of processes. We consider only well-formed conversations that satisfy the closure property. Meeting this property while adding a new participant into a well-structured conversation will simply require that the new one communicates directly with at least one existing participant.

3.1. Conversation Related Events

We describe events which a process needs to execute for engaging in a conversation. Consider a given execution C of a conversation named *c*. We will say that P_i *enters* C when it establishes a recovery point for that execution and we denote this event as $Enter_i(C)$. After $Enter_i(C)$, P_i executes the highest ordered program of its redundant software for *c*; during this execution, it exchanges application messages only with participants that have entered the execution C. If P_i sends a message to P_j that has not yet entered C, the message is assumed to be buffered at the node of P_i until P_i enters C. P_i blocks if a message it expects from P_j has not arrived. If P_j crashes, the node operating system (assumed reliable) informs this event to all processes in $Comm_j(c)$ which will treat this notification as a failure exception.

---

[1]Handling node crashes will require membership service; see section 5.1.

When P_i completes the program execution, it executes its LAT. If it fails LAT, it executes $Rollback_i(C)$: it sends a *roll-back* message to all processes in $Comm_i(c)$, restores the recovery point, and executes the next alternate program if there is one. Every participant that receives a roll-back message, executes Rollback(C) if it has not done so already. The closure property guarantees that all participants roll back.

If P_i passes the LAT for *c*, we will say P_i *exits* C and denote this event as $Exit_i(C)$. Following $Exit_i(C)$, P_i concurrently (i) looks-ahead for executing another conversation *c*' if there is one; and (ii) attempts to validate C only if all conversation executions it has previously exited have been validated. P_i's attempt to validate the execution C involves P_i sending an *ok*(C) message to all participants it knows of. When it knows that all participants of *c* have issued an *ok*(C) message, it regards the execution C to be validated and discards the recovery point for C. This event of P_i discarding the recovery point is called P_i *quitting* C and is denoted as $Quit_i(C)$.

We observe that a given conversation *c* may be executed more than once in a system run. A re-execution of *c* may be due to participants *retrying c* with alternate programs in case the previous execution(s) is rolled back. Also, the application may be so designed that processes *engage* in *c* more than once at different times during the run, with each engagement giving rise to zero or more retries at the run time. With this in mind, we model a system run to be made up of conversation executions that are sequentially numbered as C_1, C_2, C_3, C_4, .. so on. For example, C_1 may represent the first execution of a conversation named *c*, C_2 the first execution of another conversation *c*', C_3 the first execution of yet another conversation *c*", C_4 the second execution of *c*. Here C_4 can either be a retry of *c* after C_1 is rolled back or the first attempt in the second engagement of *c*. Note that if *c* and *c*' have no common participant, C_1 and C_2 may well have occurred at the same time; so, in this case, the sequential subscripting of C's does not reflect the temporal ordering of their occurrences. However, when two executions have common participants (as in C_1 and C_4), any participant that entered the higher numbered execution, did so only after entering the lower numbered execution. In what follows, subscript of C will be omitted when only one conversation execution is considered.

Let C be the n-th execution of some conversation *c*, for some $n \geq 1$. Part(C) and $Comm_i(C)$ are Part(*c*) and $Comm_i(c)$, respectively. If P_i enters C and does not crash, $Enter_i(C)$ is always followed by either $Rollback_i(C)$ or $Quit_i(C)$ (but never by both or neither); $Exit_i(C)$ never happens after $Rollback_i(C)$ and always precedes $Quit_i(C)$.

Let C_m and C_{m+1} be two consecutive executions that correspond to the first and the second (retry) executions of a conversation *c*, respectively. When P_i, $P_i \in$ Part(*c*), rolls back C_m and resumes immediately C_{m+1} using the recovery points it established for C_m, $Enter_i(C_{m+1})$ will be a null event.

3.2. Message Related Events

We consider two types of message-related events that are of interest: $Send_i(m)$ and $Deliver_i(m)$ denote the events of P_i sending m and being delivered m (for processing),

respectively[2]. $Sent_i(m)$ and $Delivery_i(m)$ are predicates which become true when $Send_i(m)$ and $Deliver_i(m)$ occur respectively.

We use Lamport's "happened before" relation denoted by '$\rightarrow$' [L78]. It is defined on system events as the smallest relation satisfying: (1) if event a and then event b occur in the same process, $a \rightarrow b$; (2) if a is $Send_i(m)$ occurred in P_i and b is $Deliver_j(m)$ occurred in P_j then $a \rightarrow b$, and (3) $a \rightarrow e$ if $a \rightarrow b$ and $b \rightarrow e$.

It is not necessary that two events are related by $\rightarrow$; such unrelated events are called *concurrent* events. Precisely, two events a and b are said to be *concurrent* if neither $a \rightarrow b$ nor $b \rightarrow a$ is true. Thus, $\rightarrow$ partially orders events. In this paper, we will apply $\rightarrow$ on the set of events defined earlier, i.e., the events related to application messages and conversation participation.

Figure 4 depicts a set of events which processes P_i and P_j could execute in a system run (suffixes i and j are omitted in the Figure). It is assumed in the Figure that C_1 is the first conversation P_i enters. Within C_1, P_i delivers (an application message) m_1 (from a participant not shown here) which is then followed by its sending of m_2 and m_3, and $Exit_i(C_1)$. P_i then looks ahead into C_2, and delivers two messages and sends one. When P_i exits C_2, C_1 is still not validated. Before $Quit_i(C_1)$, P_i enters C_3 and exits after sending m7 to P_j during that execution. P_j enters C_3, sends m_8, delivers m_7 and then exits C3. From the definition of $\rightarrow$, we can say: $Enter_i(C_1) \rightarrow Deliver_i(m_1) \rightarrow Send_i(m_7) \rightarrow Quit_i(C_1)$; $Enter_i(C_3) \rightarrow Exit_j(C_3)$; $Enter_j(C_3)$ and $Exit_i(C_3)$ are concurrent events only if m_8 is not sent to P_i.

2 We consider only application messages here. To enforce conversation rules, participants exchange control messages, such as rollback message, and these are not considered as they do not directly influence the outcome of the application level processing.

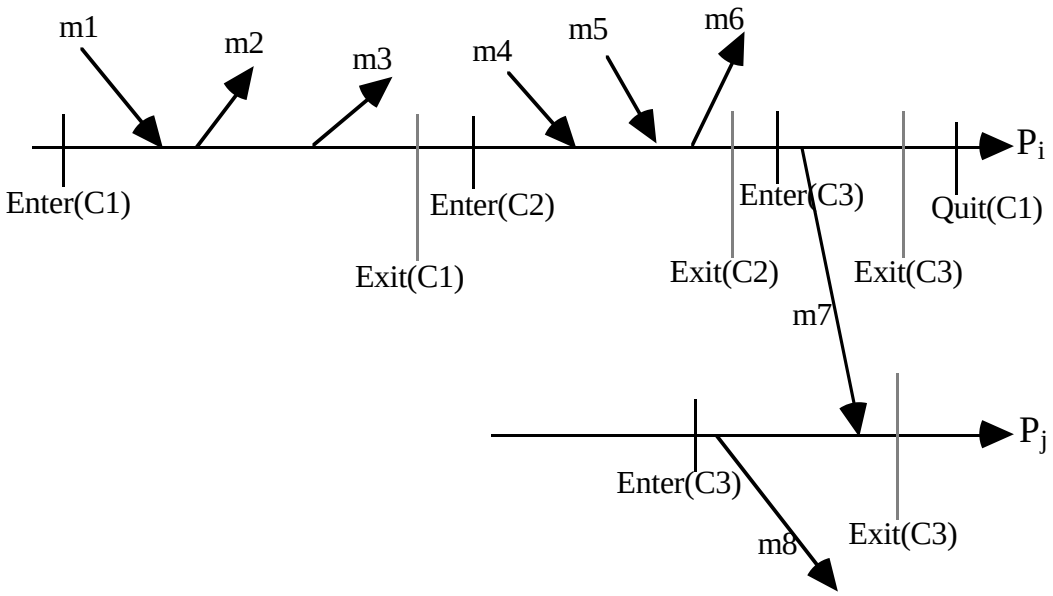


Figure 4: An example set of events.

3.3. A Pre-condition for Validation

We state the rule by which P_i judges whether it can attempt to validate C, after it has passed the LAT. The rule is that P_i disseminates an *ok*(C) message only after the conversation execution (if any) which it has entered before entering C is either known to be validated or enveloping C. The objective of this condition is to prevent P_i from sending an *ok*(C) message until it knows that it is safe for other participants of C to discard the recovery points they have established for C. Suppose that P_i has entered C_1, C_2, C_3 in that order. Let P_i, at the time of exiting C_3, have looked ahead into C_2 from C_1, and into C_3 from C_1 and C_2. Figure 4 depicts the scenario considered here. P_i in this scenario should not disseminate *ok*(C_3) until C_2 is validated (and also not disseminate *ok*(C_2) until the validation of C_1 is known). Otherwise, P_j which is not a participant of C_2, may discard its recovery point for C_3; this makes the roll back of C_3 impossible when it is necessitated by the roll back of C_2.

Suppose that C_3 is nested within C_2. This means that Part(C_3) is a subset of Part(C_2) and that every process in Part(C_3) enters C_3 after entering C_2, and exit C_3 before exiting C_2. Figure 4 will depict this case if $Exit_i(C_2)$ is moved after $Exit_i(C_3)$ and an $Enter_j(C_2)$ is introduced before $Enter_j(C_3)$. In this situation, P_i cannot disseminate *ok*(C_2) until C_1 is validated. However, it is free to disseminate *ok*(C_3) message and its validation attempts for C_3 are not restricted by whether C_1 has been validated or not. This is because, when a roll back of C_1 triggers that of C_2, rolling back C_2 will undo all the activities of nested C_3.

4. Look-ahead and Ordering of Events

In a classical conversation scheme, a participant waits for the conversation execution to be validated after it has passed the LAT. In other words, it does not send any (application) message between Exit and Quit events. Therefore, during a conversation execution C, if participants P_i and P_j execute $Exit_i(C)$ and $Exit_j(C)$ respectively, then these events are necessarily concurrent. Look-ahead however permits P_i to send or deliver m (pertaining to another conversation execution) between $Exit_j(C)$ and $Quit_j(C)$. This, as we demonstrate below, can make $Exit_i(C)$ and $Exit_j(C)$ not be concurrent, and can lead to either $Exit_i(C) \rightarrow Exit_j(C)$ or $Exit_j(C) \rightarrow Exit_i(C)$. Thus, look-ahead can give rise to different partial ordering on the same set of events, and hence to system behaviours that are never exhibited in executions with no look-ahead for the same input set.

P 3

```
enter C1;
readin r: {0,1};
send r to p1;
send((r+1) mod 2)
          to p2;
enter C2;
 ...
 deliver msg from
P4;
 if msg.value > 0 ->
    r= (r+1) mod 2;
 ...
leave C2;

send r to P1;
send((r+1) mod 2)
          to p2;
leave C1;
```

P 1

```
enter C1;
deliver r from P3;
if (r==1) ->
  send msg to P5;
  ...
  deliver r from P3;
if (r==1) ->
  send msg to p5;
leave C1;
```

P 2

```
enter C1;
deliver r from P3;
if (r==1) ->
   send msg to P5;
   ...
   deliver r from P3;
if (r==1) ->
  send msg to p5;
leave C1;
```

P 5

```
enter C1;
deliver msg from P1
             or P2;
if msg delivered from
P1   -> do A1;
if msg delivered from
P2   -> do A2;
enter C2;
  ...
  ...
  send msg to P4;
  ...
leave C2;
deliver msg from P1
             or P2;
if msg delivered from
P1  -> do A1;
if msg delivered from
P2  -> do A2;
leave C1;
```

Figure 5: An example program.

In the example of Figure 5, we consider a system of 5 processes. P_3 enters C_1 and reads the boolean input r, say, from a sensor. The value read is sent to P_1 and its complement to P_2. P_3 then enters C_2 (a nested execution). We use the term “leave” to mean Quit for classical executions and Exit for look-ahead executions. Within C_2, P_3 expects to deliver a message from another sensor process P_4. If the value contained in the delivered message is larger than zero, r is set to its complement. After leaving C_2, P_3 again sends r to P_1 and the complement of r to P_2. The code for P_1 and P_2 during C_1 are identical. Each expects to be delivered of a message from P_3 twice. Each time if the delivered value is 1 it sends a trigger message to the control process P_5. Soon after entering C_1, P_5 expects to receive a trigger message from either P_1 or P_2. If the trigger message comes from P_1, P_5 carries out the activity A_1; otherwise it performs activity

A_2. After leaving C_2, P_5 again expects to receive a trigger message from either P_1 or P_2 and carries out either the activity A_1 or activity A_2.

Figure 6 illustrates a classical execution when the value of r read by P_3 is 1 and the value supplied by P_4 to P_3 is larger than zero. m_{15} and m_{25} are two trigger messages that P_5 delivers respectively from P_1 before entering C_2 and from P_2 after quitting C_2. So, it executes A1 and then A2. Figure 7 depicts a look-ahead execution for the same input values: r=1 and the value supplied by P_4 is larger than zero. Here, P_3 is a fast process and sends m'_{31} and m'_{32} in C_1 soon after exiting but before quitting C_2. m_{15}, the first trigger message P_1 sends to P_5, is slow to reach P_5 and m_{25} is delivered first while P_5 is waiting for a trigger message either from P_1 or P_2 before it enters C_2. Consequently, it results in P_5 executing A2 and then A1. Note that P_5 is not constrained by conversation rules not to deliver m_{25}, since m_{25} was sent in C_1 and P_5 was executing C_1 when it was delivered m_{25}. This behaviour of P_5 can never happen in the classical execution for the chosen input set. Note that, in Figure 7, due to premature delivery of m_{25} i.e., $Deliver_5(m_{25})$, $Exit_3(C_2) \rightarrow Enter_5(C_2) \rightarrow Exit_5(C_2)$; that is, $Exit_3(C_2)$ and $Exit_5(C_2)$ are not concurrent.

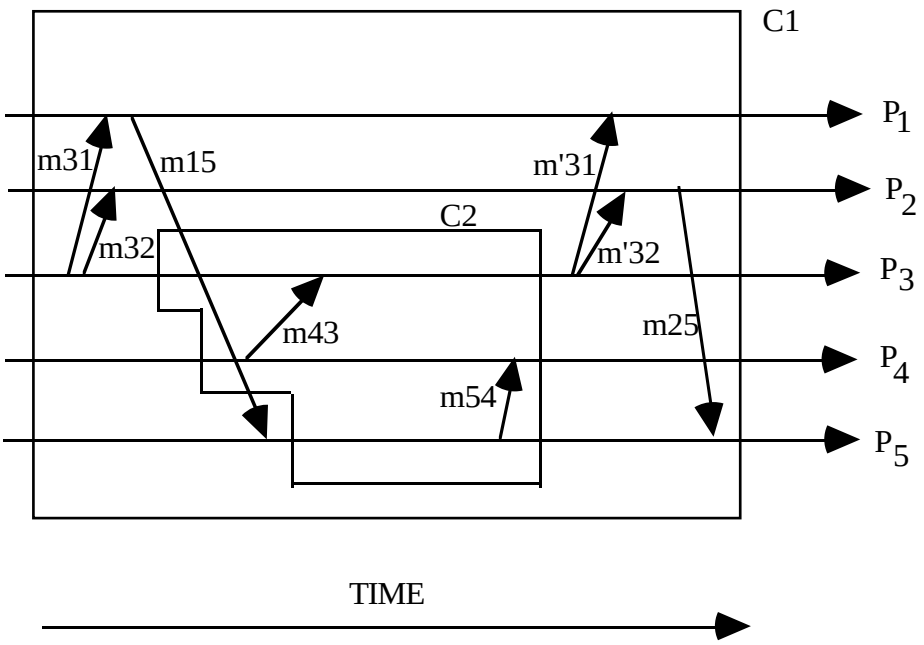


Figure 6: An execution without look-ahead.

We make two observations from Figures 6 and 7. The set of events that occurred in both the executions is same, only their ordering according to the "happened before" relation is different. This difference is solely attributed to the loosening of process synchronisation in the look-ahead execution. That is, introducing look-ahead for systems that are designed with the classical conversation rules in mind, can result in incorrect system behaviour. Secondly, P_5 delivering m_{25} before m_{15} in Figure 7 cannot be avoided even if messages are delivered to processes using a causal order delivery service (e.g. [RS91]) or even by a total order service that respects causality

(e.g. [RF96]). Such a service enforces the following message delivery order: When m and m' are sent to P_i, $Deliver_i(m) \rightarrow Deliver_i(m')$ if $Send(m) \rightarrow Send(m')$. In Figure 7, $Send_1(m_{15})$ and $Send_2(m_{25})$ are concurrent; so, a causal delivery service cannot guarantee that $Deliver_5(m_{15}) \rightarrow Deliver_5(m_{25})$. In other words, ensuring that look-ahead does not alter the system behaviour requires additional measures to be taken for preserving exit concurrency.

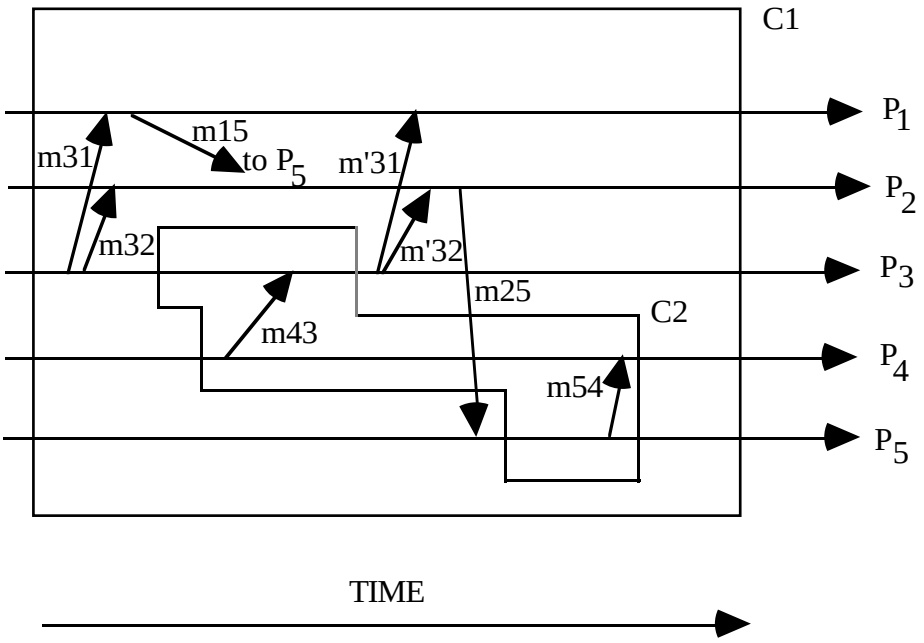


Figure 7: A look-ahead execution.

5. Preserving Exit Concurrency

Consider a message m sent (by some process) to P_j, $P_j \in Part(C)$, such that $Exit_i(C) \rightarrow Send(m)$[3]. Exit concurrency in look-ahead executions can be maintained only if m is guaranteed to be delivered to P_j only after P_j also has exited the conversation execution C. This guarantee ensures that there exists no m that can cause $Exit_i(C) \rightarrow Deliver_j(m) \rightarrow Exit_j(C)$; that is, $Exit_i(C)$ and $Exit_j(C)$ are guaranteed to be concurrent. For example, in Figure 7, $P_5 \in Part(C_2)$ and $Exit_3(C_2) \rightarrow Send_3(m'_{32}) \rightarrow Send_2(m_{25})$. If the delivery of m_{25} to P_5 had been delayed until P_5 exits C_2, P_5 would have been forced to wait for m_{15}, and the system behaviour, despite look-ahead by P_3, would be the same as in the execution with no look-ahead.

Formally, the requirement for preserving exit concurrency in look-ahead executions is:

If $P_i, P_j \in Part(C)$ and m is sent to P_j then

$$Exit_i(C) \rightarrow Send(m) \wedge Delivery_j(m) \Rightarrow Exit_j(C) \rightarrow Deliver_j(m).$$

[3]Where the identity of the process that sends m is irrelevant, we omit the subscript in Send(m).

5.1. Virtual Synchrony and Exit Concurrency

Before we present the schemes for meeting this requirement, we present a very similar requirement identified in another area of distributed computing: reliable group communication. *Virtual synchrony* [BS91] is a useful paradigm to build reliable group services and to express useful properties of system behaviour. Suppose that processes P_i and P_j are members of a group G (instead of being participants of a conversation execution C). To cope with member crashes (and also inclusion of new members), each member has a membership service that removes crashed members from (also add new members into) the membership view made available to the process. Let $Remove_i(P_k)$ be the event of P_i's membership service removing a member P_k from P_i's view for G. The main requirement for virtual synchrony is that $Exit_i(C)$ be replaced by $Remove_i(P_k)$ in the requirement identified for look-ahead executions: If P_i, $P_j \in$ Members(G) and m is sent to P_j in G then $Remove_i(P_k) \rightarrow Send(m) \wedge Delivery_j(m) \Rightarrow Remove_j(P_k) \rightarrow Deliver_j(m)$.

This similarity is not surprising. Due to variability in message delays, group members of an asynchronous distributed system can observe the crash of a member differently with respect to the ongoing message delivery. Virtual synchrony ensures that members observe a change in membership *consistently* with regard to delivery of application messages: a message whose sending happens after the removal of a member is never delivered before the delivering member has removed the crashed member. Permitting look-ahead tends to remove the process synchronisation mandated by the classical conversation rules. This asynchrony, as our example showed, can lead to a message whose sending happens after a participant exiting C, being delivered by another participant that has not even entered C. Realising the stated requirement ensures that participants observe Exit events *consistently* with respect to the messages they deliver, i.e. in the same way they would do if look-ahead had not been allowed.

5.2. Scheme 1: Piggybacking History Information

This scheme is motivated by the following observation in Figure 7. Suppose that P_3 piggybacks onto m'_{32} the information that m'_{32} was sent between $Exit_3(C_2)$ and $Quit_3(C_2)$, and also that P_2 piggybacks that information onto m_{25}. Now, P_5 can deduce that sending of m_{25} happened after a participant exiting a conversation which it has not yet entered, and therefore delay the delivery of m_{25} until it exits C_2. This will ensure exit concurrency. Thus, the scheme involves messages being piggybacked with enough history information so that a destination participant can assess the deliverability of a message. Each participant P_i performs the following four actions:

maintains a list of all C which it knows not to have been validated;

piggybacks the list to every m it sends;

a received m is not deliverable until this condition is met[4]:

for every C that is in the piggybacked list of m and $P_i \in$ Part(C),

either P_i has entered C or C is rolled back;

upon delivering m, P_i augments the piggybacked list of m, with its own list;

P_1 and P_2 learn, through m'_{31} and m'_{32} respectively, that C_2 is not yet validated. Since they are not participants of C_2, they may never learn of the validation or rolling back of C_2 when C_2 does get validated or rolled back eventually. So, processes must periodically broadcast messages informing each other about the executions that recently got validated or rolled back, so that they can keep the list size finite.

5.3. Scheme 2: Diffusing Entry Information

The second scheme is a protocol that requires each participant to send π-1 messages for a given C, where π = | Part(C) |. The protocol meets the following objective:

No participant looks-ahead of C until it knows that all participants have entered C.

Say, P_i and P_j are any two participants of C. Recall that the conversation rules dictate that a participant delivers an application message inside C only if that message is sent inside C. When P_i successfully executes the LAT of C, it is blocked by this protocol from looking ahead of C until every participant of C has entered C. Therefore, if P_i exits C and then gives rise to an application message m destined for P_j, such that $Exit_i(C) \rightarrow Send(m)$, then m will not be delivered until P_j exits C. That is, there can exist no m such that, $Exit_i(C) \rightarrow Send(m)$ and $Deliver_j(m) \rightarrow Exit_j(C)$. Thus, exit concurrency is guaranteed to be preserved when look-ahead is permitted.

The Protocol Description

In presenting the protocol, we will assume that P_i has access to a boolean variable $LAFrom_i(C)$ that is initially set to false. P_i can look ahead of C only after $LAFrom_i(C)$ becomes true and the protocol execution sets $LAFrom_i(C)$ to true once all participants of C (including P_i) are known to have entered C. We will also assume the following data structures for P_i.

$Comm_i(C)$: set of participants with which P_i is designed to communicate in C; known to P_i.

$KPart_i(C)$: P_i's knowledge of Part(C); initially set to $Comm_i(C) \cup \{ P_i \}$ if Part(C) is not known.

$EnSet_i(C)$: set of all participants known to have entered C, empty initially;

---

[4] This is in addition to the normal conversation rule that any m sent in C is delivered only in C.

$New_i(C)$: (set of) newly known participants, empty initially;

$LAFrom_i(C)$: Boolean, initially false;

$Rolledback_i(C)$: Boolean, initially false and becomes true after P_i executes $Rollback_i(C)$;

The protocol uses the following primitive for sending status messages for C:

void send_status(Status_message: Smsg, Conv-Execn-Id: C, SetOfProcesses: destn)

{ Smsg.EnSet = $EnSet_i(C)$; Smsg.KPart = $KPart_i(C)$;
Smsg.C = C; transmit Smsg to *destn*; }

The blocking primitive *void get_status(Status_message: Smsg;)* returns a received status message Smsg if there is one; it blocks, otherwise. Note that a status message should be regarded as a control message, not as an application message, since it does not directly affect the application processing. The protocol is expressed in terms of 2 concurrent threads:

Thread 1:

loop for ever

{

get-status(Smsg); C = Smsg.C; if ($Rolledback_i(C)$ == false) then
{ 1.1 if $Enset_i(C)$ does not exist then
{ create $Enset_i(C)$ = {}; create $New_i(C)$ = { }; $LAFrom_i(C)$:= false;}
1.2 $EnSet_i(C) = EnSet_i(C) \cup$ Smsg.EnSet;

1.3 $New_i(C) = New_i(C) \cup$ (Smsg.KPart - $KPart_i(C)$); }

} // end loop;

Thread 2:

upon P_i entering C :

{

2.1 if $Enset_i(C)$ does not exist then
{ create $Enset_i(C)$ = {}; create $New_i(C)$ = { }; $LAFrom_i(C)$:= false; }
2.2 $EnSet_i(C) = EnSet_i(C) \cup \{P_i\}$;

2.3 $KPart_i(C) = KPart_i(C) \cup New_i(C)$; $New_i(C)$ = { };

```
2.4     send-status(Smsg, C, KPart_i(C));}
        loop for ever   {
2.5A            New_i(C) ≠{} :  {

                        KPart_i(C) = KPart_i(C) ∪ New_i(C);
                        send-status(Smsg, C, New_i(C)); New_i(C) = { }; }

2.5B.           EnSet_i(C) == KPart_i(C) : {

                        LAFrom_i(C) = true;
                        discard Enset_i(C); discard New_i(C); quit loop; }
2.5C.           Rolledback_i(C) == true : {

                        discard LAFrom_i(C); discard Enset_i(C);
                                discard New_i(C); quit loop; }

        }  // end loop

} // end thread 2
```

The first thread collects status messages sent to P_i for any conversation execution C that P_i will enter or have entered but have not rolled-back. Whenever a Smsg for C is received, the thread carries out three sets of activities (1.1 - 1.3). First, it checks whether $Enset_i(C)$ exists; if not, the received Smsg must be the first status message received for C; so, $Enset_i(C)$ and $New_i(C)$ are created to be empty and $LAFrom_i(C)$ is set to false (1.1). Smsg.Enset indicates the participants which, according to the sender of the received Smsg, have entered C; so, these participants are added into the local $Enset_i(C)$ (1.2). Smsg.KPart indicates the participants known to the sender of the received Smsg. Those processes that are in Smsg.KPart but not in $KPart_i(C)$ are entered into $New_i(C)$, the set of newly known participants (1.3).

The second thread gets activated when P_i enters C. First the existence of $Enset_i(C)$ is checked (2.1). Non-existence of $Enset_i(C)$ means that the first thread has received no Smsg for C and P_i is the first known participant to enter C; it also leads to empty $Enset_i(C)$ and $New_i(C)$ to be created and $LAFrom_i(C)$ set to false. In (2.2) P_i enters itself into $Enset_i(C)$ and in (2.3) any new participants that the first thread may have identified are entered into $KPart_i(C)$. Then P_i sends a status message to all known participants announcing its entry into C. This is then followed by a loop that has three event-driven sets of activities (2.5A, 2.5B and 2.5C). When new participants are known, they are added into $KPart_i(C)$ and sent a Smsg (2.5A). When $Enset_i(C)$ becomes the same as $KPart_i(C)$, $LAFrom_i(C)$ is set to true and the loop is exited after $Enset_i(C)$ and $New_i(C)$ are discarded. If the thread is functioning after P_i executes $Rollback_i(C)$, it is quit after all data structures created for the protocol are discarded.

An Example

To demonstrate the working of the protocol, let us consider the following example of a C with four participants. The table below indicates the Comm set of each participant.

(Observe that the closure property is met: each participant communicates directly or transitively with every other participant.) Let us consider the protocol execution by P_1.

Process	Comm
P_1	$\{P_2\}$
P_2	$\{P_1, P_3\}$
P_3	$\{P_2, P_4\}$
P_4	$\{P_3\}$

When P_1 enters, it sends Smsg to P_2. Suppose that it has not yet received any Smsg from other participants. So, its own $EnSet_1 = \{P_1\}$ and $KPart_1 = \{P_1, P_2\}$. Now, $LAFrom_1(C)$ cannot become true until P_2 enters $Enset_1(C)$. When P_1 receives the Smsg from P_2, $EnSet_1(C) = \{P_1, P_2\}$ and $KPart_1(C) = \{P_1, P_2, P_3\}$. Note that $EnSet_1(C) \neq KPart_1(C)$; also, that P_1 and P_3 learn of each other's existence through P_2's Smsg. P_1 sends a Smsg to P_3 (in 2.5A), so does P_3 to P_1. P_3's Smsg to P_1 will have $\{P_2, P_3, P_4\} \subseteq$ Smsg.KPart; so, when P_1 receives Smsg from P_3, it learns of the participant P_4 and $KPart_1(C)$ becomes $\{P_1, P_2, P_3, P_4\} = Part(C)$. When $EnSet_1(C)$ becomes $\{P_1, P_2, P_3, P_4\}$, $LAFrom_1(C)$ is set to true after all four participants have entered C. Observe that P_1 sends 3 status messages separately during the execution; also that if P_1 had known Part(C) (i.e., $KPart_1(C) = Part(C)$) at the start of the execution, it would only carry out one broadcast to all participants. Thus, the protocol does not extract any extra message overhead for assuming that a participant need not know Part(C).

Proofs of Protocol Correctness

Theorem: Given that the participants of C neither crash nor rollback C until they complete the protocol execution, $LAFrom_i(C)$ becomes true if and only if all participants of C have entered C.

We prove this in two parts.

Lemma 1 (*Correctness*): Given that the participants of C neither crash nor rollback C until they complete the protocol execution, if $LAFrom_i(C)$ becomes true then all participants of C have entered C.

Proof: $LAFrom_i(C)$ becomes true when $EnSet_i(C) = KPart_i(C)$. Lemma is proved if we show that $EnSet_i(C) = KPart_i(C)$ becomes true only when $KPart_i(C) = Part(C)$. We prove this by contradiction, by assuming that $EnSet_i(C) = KPart_i(C)$ becomes true when a participant P_k is not in $KPart_i(C)$.

We first observe that when P_i learns of P_j's entry into C, $KPart_i(C)$ contains $Comm_j(C)$. The reasoning is as follows: P_i learns this by receiving an Smsg (with

Smsg.Enset containing P_j) from P_j itself or from some other P_m. Smsg.KPart of any Smsg from P_j includes $Comm_j(C)$; so, if P_i receives Smsg from P_j, it updates its $KPart_i(C)$ to include $Comm_j(C)$. If P_i learns of P_j's entry by receiving an Smsg from P_m, then P_m must have earlier received an Smsg with Smsg.Enset and Smsg.KPart containing P_j and $Comm_j(C)$ respectively. Therefore, Smsg.KPart of Smsg from P_m to P_i includes $Comm_j(C)$ and hence P_i updates its $KPart_i(C)$ to include $Comm_j(C)$ while it learns the entry of P_j.

Assume now that $EnSet_i(C) = KPart_i(C)$ becomes true when a participant P_k is not in $KPart_i(C)$. $P_k \in Part(C)$ and $P_k \notin Kpart_i(C)$ implies two things: P_i does not communicate directly communicate with P_k; further, by the closure property, there exists a sequence of participants P_1, P_2, ...P_l, for some $l \geq 1$, such that $P_i \in Comm_1(C) \wedge P_1 \in Comm_2(C) \wedge ... \wedge P_l \in Comm_k(C)$. Since $P_i \in Comm_1(C)$, $P_1 \in Comm_i(C)$ and hence $P_1 \in Kpart_i(C)$; also, $P_k \notin KPart_i(C)$ by assumption. So, for some x, $1 \leq x \leq l$, $KPart_i(C)$ contains P_x but not $P_{(x+1)}$, if $P_{(l+1)}$ is taken to be P_k. We can now assert that P_x that is in $KPart_i(C)$, cannot be in $EnSet_i(C)$; otherwise, the Smsg that informed P_i about entry of P_x into C will contain $Comm_x(C)$ in its Smsg.KPart and $P_{(x+1)} \in Comm_x(C)$. Therefore $P_{(x+1)}$ must be in $KPart_i(C)$. But $P_{(x+1)}$ is not in $KPart_i(C)$ by assumption. With P_x not in $EnSet_i(C)$, we cannot have $EnSet_i(C) = KPart_i(C)$. This is a contradiction.

Lemma 2 (*Liveness*): Given that the participants of C neither crash nor rollback C until they complete the protocol execution, if all participants of C enter C then $LAFrom_i(C)$ becomes true.

Proof: The arguments of Lemma 1 show that $LAFrom_i(C)$ cannot become true with P_k, $P_k \in Part(C)$ and $P_k \notin Kpart_i(C)$. Thus, the second thread runs until $Kpart_i(C) = Part(C)$. Observe that every Smsg sent to P_i is received and processed by the first thread; further, a participant sends a Smsg to every participant it knows. Therefore, $EnSet_i(C) = Kpart_i(C)$ becomes true when every participant enters C and sends Smsg to P_i.

6. Conclusions

This paper investigates the effects of using "classical" conversation schemes in the distributed systems and of relaxing the synchronisation required by the schemes. Kim's look-ahead conversation scheme relaxes the synchronisation requirement on conversation exit and thereby provides scope to enhance the system performance. The analysis presented in the paper demonstrates that an execution with look-ahead can produce system behaviours that are never exhibited in an execution without look-ahead. We identified the cause of this anomaly to be the absence of exit concurrency. We then proposed two simple schemes to preserve this exit concurrency. The proposed schemes do not require a conversation participant to know a priori all participants of the conversation, but only those with which it is designed to interact during the conversation.

We have not formally shown that preserving exit concurrency is also the sufficient condition for executions with and without lookahead to be equivalent. This will be our future work. For now, we informally argue that the exit concurrency is also sufficient, based on intuitive reasoning: the only effect look-ahead has on a conversation execution is to allow participants to quit the conversations at widely different physical times. When exit concurrency is maintained, those quit events take place logically at the same time, in relation to the application messages delivered by participants. So the net effect is equivalent to what would have happened if look-ahead had not been allowed.

Acknowledgements

The work described here has been supported in part by DeVa (Design for Validation) ESPRIT Project and by EPSRC/UK DISCS Project. Comments and criticisms of Professor B.Randell and Dr. R.Stroud are acknowledged.

References

[BS91] K. Birman, A. Schiper, P. Stephenson. Lightweight Causal and Atomic Group Multicast. ACM Trans. on Computer Systems, 9, 3, 1991, pp. 272-314.

[G78] J. N. Gray, “Notes on database operating systems”, *Lecture Notes in Computer Science*, Vol. 60, Springer Verlag, 1978, pp. 393-481.

[KB97] K.H. Kim, L. Bacellar. Time-Bounded Cooperative Recovery with Distributed Real-Time Conversation Scheme. Proc. Third IEEE Workshop on Object-Oriented Real-time Dependable Systems (WORDS'97), Newport Beach, California, February 1997.

[KY89] K.H. Kim, S.M. Yang. Performance Impacts on Look-Ahead Execution in the Conversation Scheme. IEEE TC-38, 8, 1989, pp. 1188-1202.

[L78] L. Lamport. Time, Clocks, and Ordering of Events in a Distributed System. CACM, 21, 7, 1978, pp. 558-565.

[R75] B. Randell. System Structure for Software Fault Tolerance. IEEE TSE-1, 2, 1975, pp. 220-232.

[RS91] M. Raynal, A. Schiper, S. Toueg. The causal ordering abstraction and a simple way to implement it. Information Processing Letters, 39, 1991, pp. 343-350.

[RF96] L.E.T. Rodriques, H. Fonseca, P. Verissimo. Totally Ordered Multicast in Large-Scale Systems. Proc. 16th Int. Conf. on Distributed Computing Systems. 1996, pp. 503-510.

[XR98] J. Xu, A. Romanovsky, A. Zorzo, B. Randell, R.J. Stroud, E. Canver. Developing Control Software for Production Cell II: Failure Analysis and System Design Using CA Actions. DeVa ESPRIT LTR Project. Third year deliverable. 1998 (submitted to FTCS-29).

[YK92] S.M. Yang, K.H. Kim. Implementation of the Conversation Scheme in Massage-Based Distributed Computer Systems. IEEE TPDS-3, 5, 1992, pp. 555-572.

[ZR98] F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R.J. Stroud, I.S. Welch. Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study. Software: Practice & Experience. 1998 (accepted).