An Efficient Algorithm for Multi-Process Shared Events [1]

Vijay K. Garg[2]
Department of Electrical and Computer Engineering,
University of Texas, Austin, TX 78712-1084
email: vijay@pine.ece.utexas.edu

Abstract

Many problems in distributed computing systems require execution of events shared by multiple processes. In this paper, a fair and efficient algorithm for multi-process shared events is presented. We also present its application to distributed implementation of generalized CSP alternative command. We show that our algorithm is simpler and has lower message and time complexity than proposed implementations for generalized CSP alternative command, and distributed algorithms for N-party interactions.

1. Introduction

Wide availability of computer networks, and low cost of hardware has made it desirable to use distributed systems. Distributed systems, however, are difficult to design and often need tricky synchronization between multiple processes. The synchronization is required to coordinate multiple processes for events, which must be executed either by all, or none of them. Some examples of such shared events are distributed transactions in databases that require commit by either all or none of the processes, and atomic broadcasts that require that a message be received by either all or none of the processes. Shared event, or a multi-party interaction, is such a useful concept that it also appears in Coupled State Machines [Bochman 80], Petri Nets [Peterson 81], CSP [Hoare 85], CCS [Milner 80], RADDLE [Forman 86], and ConC[Garg 91]. Ease in specification of concurrent systems using shared events also provides a strong motivation for the search of an efficient algorithm for its execution.

The problem of execution of multi-process shared events can be described as follows. There are n geographically distributed processes. Each process is either idle or executing. Each process when idle is willing to execute any event that is enabled in its current state. Any algorithm for this problem is required to coordinate the processes for execution of events, such that no process is asked to participate in more than one interaction. The algorithm should also be free from deadlock and starvation. We assume that processes communicate with each other by means of asynchronous reliable messages. The committee coordination problem [Chandy 88] is a special case of this problem where the same events are enabled for all waiting sessions.

---

[1]A preliminary version of this paper was presented in Proc. 2nd IEEE Symposium on Parallel and Distributed Processing, Dallas, TX 1990

[2]research supported in part by NSF Grant CCR 9110605, TRW faculty development award, and IBM Agreement 153.

As an example of a multi-party interaction, consider the distributed players problem. Assume that there are four players who are interested in playing chess, tennis, poker, and bridge. Joe plays only chess, bridge and tennis. Mary plays all four games, while Jack and Bob play only bridge and poker. There are also some constraints in the execution of these events. Joe will play only tennis after chess. Similarly, Mary plays only poker or tennis, if she played chess, or bridge last. Since games require cooperation between two or more players the players may have to wait for each other. Also the players are in different cities (i.e. on different processors), and can communicate only through mail (asynchronous messages).

Our algorithm assumes that all messages sent by one machine to another are received uncorrupted in FIFO order. A service can easily be provided by a communication protocol layer that detects duplicate, lost, out-of-sequence and corrupt messages. We also assume that local and global causality as proposed by Lamport [Lamport 78] is preserved by clocks of various machines. This can easily be provided by an extra layer of clock synchronization that uses Lamport's algorithm.

This paper is divided into five sections. Section 2 describes the related work in execution of shared events. Section 3 presents the algorithm. Section 4 discusses its correctness, and its message and time complexity. Section 5 describes some efficiency considerations in implementation of the algorithm.

2. Related Work

The execution of shared events also arises in implementation of the generalized I/O command of CSP. A CSP program, as described in [Buckley 83], consists of a set of processes that communicate with each other using synchronized message passing. Communication between processes occur when two processes have matching input and output statements. The alternative command of CSP provides non-determinism by letting a process select one of the several statements for processing. Each statement is protected by a guard (a boolean expression and/or one input statement) which must be enabled for the statement to be considered for selection. A guard is enabled if the boolean expression evaluates to true and the named output process has not terminated. However, not all algorithms are easy to express using only the constructs of CSP. Researchers have found it useful to extend the notion of guard to include output command and many implementations have been presented [Buckley 83, Bagrodia 86, Ramesh 87, Lee 87]. This generalized CSP construct is obviously a special case of multi-process synchronous events problem.

[Buckley 83] presented four conditions that should be satisfied by an effective implementation of the CSP I/O construct. These conditions are minimality of the processes involved, minimality of system information at each process, low number of messages required and the selection of a ready interaction in a bounded amount of time. They showed that [Silberschatz 79, Snepscheut 81] did not satisfy one or more of these conditions. [Back 84] improved upon this result by providing an implementation that satisfied two more conditions: weak fairness and bounded time for a process to determine if it can communicate with some process. [Ramesh 87] provided an improved implementation which could be extended to allow multi-process synchronization. [Bagrodia 85, 88, Levy 88, Kumar 90] also describe algorithms for this purpose. We provide a new distributed algorithm that satisfies all six conditions, and is extensible to the case of multi-process synchronization. It is simpler than algorithms presented in literature. It differs from earlier algorithms in the following ways:

- **Sequential Capturing**: In [Ramesh 87, Bagrodia 88, Kumar 90], a process captures all processes participating in an event sequentially to avoid any deadlock. This can result in a substantial delay for events that is shared by a large number of processes. The time complexity of such algorithms is dependent on the number of participating processes. Since processes are captured for a long time it also means that other processes may have to wait for a long time for captured processes to be released. In our algorithm, a process tries to capture all participating processes in parallel.

- **Message Load**: In [Ramesh 87, Kumar 90], each process attempts to captures other processes involved in the desired interaction. To avoid deadlocks, processes are required to be captured using a static ordering between them. This may result in swamping of high-priority processes with many messages. Our algorithm can distribute the message load more evenly as it does not depend upon any static ordering. This aspect is discussed further in Section 5. Moreover, in the algorithm proposed by [Kumar 89] the ***trying*** message includes a sequence of processes that the token has visited in the current round which increases the information transferred to each participating process. Thus, the length of the message would increase as the number of participating processes increases.

- **Fairness in Execution**: In [Ramesh 87], a guard is chosen at random by each process, thereby guaranteeing that any guard has finite probability of being chosen. This approach, however, does not provide any bound on the number of events a guard may have to wait before it is executed. We provide such a bound by ensuring that no interaction is aborted in favor of some other interaction more than once.

- **Timestamped Messages**: [Ramesh 87, Bagrodia 88, Kumar 89] do not use timestamps in their algorithms. Our algorithm requires global causality of timestamps, and we assume that there is a clock synchronization algorithm such as proposed by Lamport[Lamport 78] running on the network. Since Lamport's algorithm is very simple to implement and does not incur high penalty, this is a not a serious drawback of our algorithm. Due to usefulness of global causality, other algorithms may already be using a clock synchronization algorithm.

3. Description of the Algorithm

The algorithm is event-driven and can be informally described as follows. Each interaction is assigned a master. A master can execute an interaction, if all participating processes commit to it. A process can commit to an interaction by sending a ***yes*** message to its master. A master requests for these messages by means of ***request*** messages. A ***request*** message may either be delayed, or be responded with a ***yes/no*** message. If all the participating processes commit, then the master sends a ***success*** message to them. On receiving a ***success*** message, a process can execute the interaction.

When a process is in execution state (executing some interaction), it responds to only two kinds of messages - ***ready*** and ***request***. On receiving a ***ready*** message, it makes a note in its ready table, whereas a ***request*** message is replied by a ***no***.

Once a process comes to an alternative command, it sends ***ready*** message to all the masters for the guards it is ready to execute. Then, if any interaction is ready, it sends out ***request*** messages. After this the process takes any action only on receiving a message.

Some of the features of this algorithm are as follows:

1. A process can send a ***yes*** message to at most one master. A process that has committed to an interaction can not send ***request*** message for any other interaction. Thus, a process commits to only one interaction. This way of committing resembles two-phase commit protocol used in databases for implementing transactions. The difference between two problems is, that in databases if two transactions t_1 and t_2 are eligible at some state, then the protocol needs to ensure that the final execution can be written either as t_1t_2 or t_2t_1. In this problem, once t_1 is executed t_2 may not be valid any more. For example, initially both tennis and chess may be eligible, but once tennis is played chess may not be eligible any more. The notion of fairness is also not so important for database applications.

2. A process that has already committed, on receiving a request for another interaction, says ***no*** to a younger process and delays the older process. If the process is the master of its committed interaction, and the interaction has not received all the ***yes*** messages, then it is aborted in favor of an older interaction. This way there cannot be any deadlock between different interactions. This strategy is commonly referred as wait-die strategy in databases[Eswaran 76].

3. The fairness is based on the principle that if there is a choice in execution of an interaction, then the interaction which has waited for a longer time is chosen. With each request message for an interaction, the master also sends the timestamp of its last execution.

The algorithm as shown in Figure 1 uses the following messages:

ready sent by a process to the master of an interaction indicating its willingness to execute the interaction This message can be sent to multiple masters. They increase the probability that a ***request*** message succeeds.

request sent by the master to processes for the ***yes/no*** reply, With a request message, the master also sends the time when it was executed last.

yes sent by a process to the master of an interaction indicating its willingness to execute the interaction. This message is sent to only one master.

no sent by a process to the master of an interaction indicating that it has committed for some other interaction

success sent by the master to processes asking them to execute the interaction

abort sent by the master to processes asking them to abort the interaction

```
Background()
   if (mtype = ready) update(ready_table)
         else if (mtype = request) reply(currmess, no);

Initialize()
   captured = 0; delayed[ ]=0; initialize_guards;
   send ready messages to various masters;
   if any interaction is ready then sendrequest(myinteraction);

Handle_Ready()
   update(ready_table);
   if (the interaction is ready) and
   (I am not exploring any other interaction) then sendrequest;

Handle_Request()
   if (guard[interaction]=closed) reply(currmess, no);
   else if (myinteraction = 0) /* I am not committed */
         myinteraction:=interaction;
         reply(currmess, yes);
   else if (timestamp[interaction] > timestamp[captured])
         reply(currmess, no);
   else if (master[captured] = myid)
         sendabort(myinteraction);
         myinteraction = interaction;
         reply(currmess, yes);
   else delayed[currmess.src]=interaction;

Handle_Abort()
   try_another_interaction;

Handle_Succ()
   if (captured = currmess.interaction)
         takeinteraction( currmess.interaction);

Handle_Yes()
   checklist[currmess.src]=0;
   if all_have_responded_yes
         takeinteraction(currmess.interaction);
         sendsucc(currmess.interaction);

Handle_No()
   rstatus[interaction][src] = false;
```

sendabort(interaction);
try_another_interaction;

try_another_interaction()
if any process delayed respond to it;
else if any interaction ready send request
else send ready to masters which have been sent no

Figure 1: Algorithm for Execution of Multi-process Events

Some of the data structures are as follows:

ready_table: a table maintained by the master of an interaction. If a participant of a handshake sends a ready message to the master, he is marked as ready in a table. For performance reasons, we do not require processes to send "not-ready" messages when they are not ready for a handshake. Thus, an entry in a ready table is only one-way correct. If it indicates that a process is not ready for a handshake, then this is true for the steady state of the system. However, if it says that a process is ready for some handshake, then this must be confirmed by a request message.

guard[interaction]: Is the interaction enabled in my current state?

captured: the interaction that has captured me.

4. Correctness of the algorithm

In this section, we informally prove that the algorithm shown in Figure 1 is correct. We show that the algorithm is safe, that is it can ask a process to participate in at most one interaction. We also show that the algorithm is live, that is if one or more interactions are enabled, the system will execute some interaction.

Theorem 1: *Each process can be asked to participate in at most one interaction.*

Proof: A process commits for an interaction only if it has sent ***yes*** in response for the interaction or it is master for that interaction and has sent ***request*** messages. Since a process can have at most one outstanding ***yes*** message if it has not sent out any ***request*** message, and none if it has sent one, a process cannot commit for two interactions. Q.E.D.

Theorem 2: *If one or more interactions are eligible then the system will execute an interaction.*

Proof: Consider the master of the interaction with the oldest timestamp. When this master sends out the ***request*** message, if all processes respond with *yes*, the interaction can be executed. Since the interaction is eligible, and it has the highest priority, no process can send *no* for the interaction. The only other option for them is to delay their response.

We define the delay graph D as a directed graph $D = (V, E)$ where V is the set of all the processes. There is a directed edge from process v_1 to v_2 if there exists a process that has delayed v_1 in favor of v_2. This implies that the priority of v_1 is greater than v_2 because we delay the older process. Global causality implies that the graph is acyclic. We traverse the path of processes in the delay graph. Since the delay graph is acyclic, we will reach a node which has no outgoing edge. This process being youngest will receive answer from all the processes and therefore can send *success/abort* message to all its processes in its set which then can reply to their delayed masters. If the decision was *abort* then the path delay graph has less number of edges and this particular interaction will not be explored again. If the decision was *success*, an interaction is executed. Q.E.D.

For example, consider the example of distributed players. Let Joe be the master of tennis, Mary of chess, Bob of poker and Jack of bridge. Let the last time tennis was played be 12, chess be 15, poker be 14, and bridge be 18. Assume that tennis is enabled because all participating players are willing to play it. Also assume the Bob and Mary are willing to play poker but Jack is not. The following event sequence describes a typical scenario:

(1) Bob sends a ***request*** to Mary for poker who responds ***yes*** as she has not committed to any other game.

(2) Joe sends a ***request*** for tennis to Mary who delays the response to this message.

(3) Bob sends a ***request*** for poker to Jack who responds with a ***no*** message.

(4) Bob sends an ***abort*** for poker to Mary, who now can respond to the delayed request of Joe.

(5) Mary can now send a ***success*** message to Joe, who then can execute the interaction.

Message and Time Complexity

The Worst Case Message Complexity

We count the number of messages a process has to handle in the worst case before it is guaranteed to succeed. We first count the number of times an interaction can abort. By our fairness rule, an interaction can be aborted in favor of some other interaction at most once. Thus, if there are p interactions, then an interaction of any process must succeed after $p-1$ or less number of attempts. Hence, the number of requests to a process for a guard is less than or equal to $p-1$, and correspondingly the number of aborts is less than or equal to $p-2$. There is at most one success message. Therefore, the number of messages in the worst case for a master guard with d slaves to succeed in executing a interaction is:

ready messages at most $(p-1)d$ in number
request message at most $(p-1)d$ in number
yes/no at most $(p-1)d$ in number
abort at most $(p-2)d$ in number
succeed at most d
Total: $4(p-1)d$ messages

The Best Case Message Complexity

In the best case, there will be no aborts; therefore, a master guard will be successful in $4d$ messages. A slave guard will require four messages for successful execution of an interaction.

Time Complexity

A major advantage of the algorithm is its time complexity. This is due to the fact that the master tries to capture all participating processes in parallel as opposed to the sequential capturing proposed by [Ramesh 87],[Bagrodia 85] and [Kumar 89]. Let t be the average message transfer time. Assuming that the computation time is negligible in comparison to the message transfer time, in the best case the time taken for a successful interaction would be $4t$ and for the worst case it would be only $4(p-1)t$. The number of processes participating in the interaction does not affect the performance the algorithm which makes this algorithm more suitable for synchronizing a large number of processes for a shared event. The performance of sequential capturing algorithms degrades as the number of participating processes increases. Let there be d processes participating in the interaction. Even in the best case each message would take t time, and so the time complexity for a sequential capturing algorithm would be $2dt$. Therefore, for a large number of participating processes even the best case performance would be quite slow.

Effective Implementation

Theorem 3: *The algorithm satisfies the following six criteria of effective implementation:*
(1) The number of processes that are involved in the selection of a guard should be minimum.
(2) The amount of system information that each of these processes should be low.
(3) When an interaction is ready then it will be selected within a finite time.
(4) The number of messages exchanged for making a selection by any process is small.
(5) The time it takes for a process to determine whether it can establish communication with some other process should be bounded.
(6) If a process has a guarded command that is infinitely often enabled, then it should eventually succeed.

Proof: (1) and (2) are obvious from the algorithm. (3), (4) and (5) follows from Theorem 2 and the message complexity analysis. (6) follows from our fairness conditions. Q.E.D.

5. Efficiency Considerations

In the above algorithm, we did not discuss how we chose masters for each interaction. The efficiency of the algorithm is dependent on this choice. We discuss some desirable requirements for the choice, and strategies to assign masters based on the requirements.

The algorithm, as presented above, requires the master of the interaction to deal with more messages than dealt by other participating processes. To prevent any machine from getting overloaded, we may choose masters such that the maximum load on any machine is minimized. The problem can be stated formally as follows: Let M and I represent the set of machines and the set of interactions respectively. Let the degree of an interaction i be the number of machines which participate in it. Our problem is to find an assignment of master for interactions, $f : I \to M$, such that the maximum load on any machine is minimized. The load of a machine is defined as the sum of degrees of all interactions for which it acts as a master. For example, in the distributed player example, the degrees of chess, tennis, poker, and bridge are 2, 2,3, and 4 respectively. A possible master assignment is as follows: Mary is the master for chess and poker, Joe is the master for tennis, and Jack is the master for bridge. The maximum load in this assignment is on Mary who has the load of Chess (2) and Poker (3). If Bob is assigned as the master of Poker, then Jack will have the maximum load of Bridge (4).

Theorem 4: Let there be m machines and n interactions. There exists an algorithm with $O(log(mn)m^2n^2)$ time to find the master assignment $f : I \to M$, such that the maximum load on any machine is minimized.

Proof: We consider a related problem which seeks the assignment of masters such that the maximum load on any machine is less than K. Let the total load (the sum of degree

Figure 2: Minimizing the Maximum Load

of interactions) be S.

[htbp]

This problem can be solved as feasible circulation in a network with upper as well as lower bounds on the capacity of each edge. We add a pseudo source s and a pseudo sink t with the following bounds:

$$l(s,h_i) = u(s,h_i) = l(h_i,m_j) = u(h_i,m_j) = degree(h_i)$$

$$l(m_i,t) = 0; u(m_i,t) = K, l(t,s) = S, u(t,s) = S \quad \forall\ h_i \in I,\ m_j \in M$$

Figure 2 shows the assignments to various edges. Using Out-of-Kilter method[Lawler 76], this problem can be solved in $O(m^2n^2)$ where m is the number of machines and n is the number of interactions. Using the solution to decision problem, we can solve the minimization problem in $O(log(mn))$ time using a binary search. Thus, the problem of finding master assignment such that the maximum load on any machine is minimized can be solved in $O(log(mn)m^2n^2)$.

6. Conclusions

Execution of shared events arises in many contexts in distributed systems. We have proposed an efficient algorithm of the shared events in a distributed environment. We have implemented this algorithm on a SUN cluster for distributed execution of decomposed Petri nets [Garg 88,92]. It is also applicable for implementation of the generalized CSP I/O command. Our algorithm is conceptually simpler and more efficient than existing algorithms.

7. References

[Back 84] R.J.R. Back, P. Eklund, and R. Kurki-Suonia, "A fair and efficient implementation of CSP with output guards", Tech. Report No. 38, Abo Akademi, Finland, 1984.

[**Bernstein 80**] A.J. Bernstein, "Output Guards and Non-Determinism in CSP", ACM Toplas, 2(2), April 1980, pp 234-238.
[**Bagrodia 85**] Rajive Bagrodia, "A Distributed Algorithm for N-Party Interactions", MCC Technical Report STP-053-85, August 1985.
[**Bagrodia 86**] Rajive Bagrodia, "A Distributed Algorithm to Implement the Generalized alternative command of CSP", Proc. of International Conference on Distributed Computing Systems (ICDCS), pp 422-427, 1986.
[**Bagrodia 88**] Rajive Bagrodia, "Process Synchronization: Design and Performance Evaluation of Distributed Algorithms ", ***IEEE Trans. on Software Engg.***, 15(9), pp. 1053-1065, Sept 1989.
[**Buckley 83**] G.N.Buckley, A.Silberschatz, "An Effective Implementation for the Generalized Input-Output Construct of CSP", ACM transactions on programming languages and systems (TOPLAS), April 1983.
[**Bochmann 80**] G.v.Bochmann, P. Merlin, "On the construction of communication protocols", in Proc. Inter. Conference on Computer Communication, 1980.
[**Eswaran 76**] K.P.Eswaran, J.N.Gray, et.al.,"The Notion of Consistency and Predicate Locks in a Database System", Comm. ACM, 18(11), Nov 1976, pp 624-633.
[**Forman 86**] I.R.Forman, "On the Design of Large Distributed Systems", Proc. International Conference on Computer Languages, 1986.
[**Garg 88**] V.K. Garg, "Specification and Analysis of Distributed Systems with a Large number of processes", Ph.D. Dissertation, Computer Science Division, University of California, Berkeley, 1988.
[**Garg 91**] V. K. Garg, C.V. Ramamoorthy, "ConC: A Language for Concurrent Programming", ***Computer Languages Journal*** Vol. 16, No. 1, January 1991 pp 5-18.
[**Garg 92**] V. K. Garg, M.T. Raghunath "Concurrent Regular Expressions and their Relationship to Petri Net Languages," ***Theoretical Computer Science*** 96 (1992) pp 285-304.
[**Hoare 85**] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1985.
[**Kumar 89**] Devendra Kumar, "N-Party Synchronization in Distributed Systems", Proc. First Annual IEEE Symposium on Parallel and Distributed Processing, May 22-23 1989, Dallas, TX, pp 397-404.
[**Kumar 90**] Devendra Kumar, "An Implementation of N-Party Synchronization Using Tokens", Proc. 10th Intl. Conference on Distributed Computing Systems, May 28-June 1, Paris, France, pp. 320-327, 1990.
[**Lamport 78**] L.Lamport, "Time, Clocks and Ordering of Events in a Distributed System" Comm. ACM, 21(7), July 1978, pp 558-565.
[**Lee 87**] I. Lee, S.B. Davidson, "Generalized I/O with Timing Constraints", Proc. of International Conference on Distributed Computing Systems (ICDCS), pp 316-323, 1987.
[**Milner 80**] A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol 92, Springer-Verlag 1980.
[**Chandy 88**] K.M.Chandy, Jayadev Misra, Parallel Program Design : A Foundation, Addison-Wesley Publishing Company, Inc., August 1988.
[**Peterson 81**] J. Peterson, Petri-Net Theory and Modeling of Systems, Prentice Hall, Inc., Englewood Cliffs, New Jersey 1981.

[Ramesh 86] S.Ramesh, "Programming with Shared Actions: A methodology for developing Distributed Programs", Ph.D. Dissertation, IIT Bombay, India, June, 1986.
[Ramesh 87] S.Ramesh, "An Efficient Implementation of CSP with Output Guards", Proc. of International Conference on Distributed Computing, 1987.
[Reif 84] J. H. Reif, P.G. Spirakis, "Real-Time Synchronization of Interprocess Communications", ACM transactions on programming languages and systems (TOPLAS), pp 215-238, April 1984.
[Reisig 85] W. Reisig, Petri Nets, An Introduction, lecture notes in Computer Science, Springer-Verlag, 1985.
[Silberschatz 79] A. Silberschatz, "Communication and Synchronization in Distributed Systems", IEEE Transactions Software Eng.-5,6 Nov. 1979, pp 542-546.
[Snepscheut 81] J.L.A. Van de Snepscheut, "Synchronous Communication between asynchronous components", Information Processing Letters, 13, 3 Dec. 1981, pp 127-130.