

On Restructuring Distributed Algorithms for Mobile Computing

R.K. Ghosh¹ and Hrushikesh Mohanty²

¹ Department of Computer Science & Engineering,
Indian Institute of Technology, Kanpur
`rkg@cse.iitk.ac.in`

² Department of Computer and Information Sciences,
University of Hyderabad, Hyderabad – 500 053
`hmcs@uohyd.ernet.in`

Abstract. In this paper we propose a formalization of a subclass of distributed algorithms that can be suitable for execution on mobile environment. In particular, our investigation puts stress on evolving techniques to include the design abstractions of the resources and synchronization typical to a mobile environment with the conventional design of distributed algorithm, so that a greater amount of concurrency can be derived within the available level of resources. To that effect we propose a method to evaluate a *concurrency load* for any abstraction and use it to come up with answer as to which of the abstractions would be better when host mobility is allowed.

1 Introduction

Until recently, distributed algorithms are designed only for a network of static hosts. Such algorithms execute by exchange of messages over the network. Most of the ideal assumptions regarding the unlimited amount of resources availability and the robustness of communication interface in designing normal distributed algorithms can not be extended to a mobile computing environment [1]. Therefore, restructuring the existing distributed algorithms for execution in mobile environment is, indeed, a great challenge. Nevertheless, certain generic observations may, to great extent, help in restructuring the existing distributed algorithms for execution on mobile environment.

1. The distributed algorithms that fragment the problem into smaller parts and attempt to execute those parts independently on many hosts to get benefit of collective computing power of many hosts are ruled out as a mobile host (MH) is resource poor. Performing heavy local computations on an MH is infeasible.
2. Some algorithms may not have any requirement for heavy computation. But may require the hosts to execute operations using globally available resources that require synchronization. This class of algorithms would be ideally suitable for restructuring in a mobile environment; as the mobile host may primarily be used to access services rather than to run services. One of such project is the Wireless Coyote [5].

In this paper, we restrict ourselves the class of distributed algorithms which do not run distributed services but those which access a service from other hosts, and, therefore, require to deal with synchronization issues. A typical example of such a service may be a distributed file system. We examine applicability of Badrinath and Acharya's two-tier approach [2] and also discuss about other generic strategies to restructure such distributed algorithms for mobile computing environment.

2 Bandwidth and Resources

A distributed algorithm comprises of a *communication* component and a *computation* component. The computation component is bound to the individual hosts. From time to time a host might communicate with its neighbours or other nodes to exchange information regarding the state of its own or other's registers, the results of current computations, contacting some hosts for ordering tasks in the algorithm or to get initialization parameters for the next phase of computation, and so on. Thus a distributed system of hosts shows bursts of communication in between periods of local computation. When hosts become mobile not only the cost of computational resources at mobile hosts increase, but two additional cost parameters, namely, *location update* and *lookup* become pronounced. The focus of this section is to examine strategies to minimize communication overheads and resource requirements by ensuring that the computation is largely restricted to the fixed part of the network.

2.1 Two-Tier Approach

The *two-tier principle* [2] states:

Principle 1 *To the extent possible, computation and communication costs of an algorithm is borne by the static part of the network. This attempts to avoid locating a mobile participant and lowers the search cost incurred by the algorithm; additionally, the number of operations performed at the mobile hosts is kept to a minimum, and thereby, consumption of battery power, which is a critical resource for mobile hosts, is restricted.*

In order to restrict the costs of location maintenance and lookup, three different strategies, namely, the *search*, *inform* and *proxy* were proposed by Badrinath and Acharya [2] in the context of restructuring the distributed token ring algorithm. These techniques are quite general and can be used in various other classes of distributed algorithms.

Instead of focusing directly on the resource and bandwidth issues, our attempt here is to examine the restructuring problem according to the systemic roles of individual hosts for implementing different distributed algorithms.

2.2 Non-coordinator Systems with All Machines Equivalent

All machines in a non-coordinated system are equivalent. The amount of computational and communication load shared by each machine is same. Such systems can easily be modified to work in a mobile computing environment. We illustrate the method with Lamport's Bakery algorithm [6] for mutual exclusion as an example.

In the Bakery algorithm, a process waiting to enter a critical section chooses a number. This number must be greater than all other numbers currently in use. There is a global shared array of current numbers for each process. The entering process checks all other processes sequentially, and waits for each one which has a lower number. Ties are resolved using process IDs. If no restructuring is done, and the algorithm is directly executed on a collection of N mobile hosts, the overall communication cost (see [4] for details) would be

$$3 * (N - 1) * (2 * C_{wireless} + C_{search}).$$

Now suppose we modify the original algorithm as follows:

1. Time stamping is used only for messages exchanged between the MSSs – which now form the components of bakery algorithm – and not for messages exchanged between the mobile hosts.
2. A mobile host initiates the algorithm by sending an initialization message to its local MSS. The MSS executes the algorithm on behalf of the mobile host. Thus the appropriate request, reply and release messages are marked with the id of the mobile host for which it is sent by the MSS.
3. When the MSS secures mutual exclusion for the MH, it informs the MH. But the mobile host might have changed its location in the meantime. Therefore, a search may potentially be required.
4. The mobile host then accesses the critical section and informs the MSS after need for access is over. A release message is sent to every other MSS.
5. If the mobile host disconnects prior to the requested access to the critical section, then every disconnection has to be informed to the MSS. On receiving the grant for access, the MSS can instantly initiate a release response in this case. Thus, other requests are not delayed.

Those interested to know the correctness of the algorithms may refer to [4] for details. The breakups for the communication cost of this modified algorithm will be as follows:

- Initialization = $C_{wireless}$,
- Granting a request = $C_{wireless} + C_{search}$, and
- Release of lock = $C_{wireless} + C_{fixed}$.

The overall cost along with the cost for executing the algorithm between MSSs is, thus,

$$3 * C_{wireless} + C_{fixed} + C_{search} + 3 * (M - 1) * C_{fixed},$$

where M is the number of MSSs participating in the algorithm execution. Since $C_{search} > C_{fixed}$ and $N \gg M$, the overall communication cost is much cheaper compared to the non-restructured algorithm.

2.3 Non-coordinated Systems with Exception Machines

Non-coordinated systems differ only in the sense that the code executed by one machine (or a few of them) is different than that executed by the most of the other machines. The machines executing different code constitute the set of *exception* machines. An example of this category is Dijkstra's self stabilizing algorithm [3].

Consider a system consisting of a set of n finite state machines connected in the form of a ring, with a circulating *privilege* which denotes the ability of a machine to change its state. when a machine has the privilege, it is able to change its current state which is termed a *move*. Let us now look at Dijkstra's algorithm involving K states, where $K \geq n$, and n is the total number machines. For any machine, we use the symbols S , L , R to denote its own state, the state of its left neighbour, and the state of its right neighbour respectively.

The exception machine, machine 0: if $L = S$ then $S = (S + 1) \bmod K$.

Other machines: if $L \neq S$ then $S = L$.

Dijkstra also three proposed a better algorithm involving three state using two exception machines. The three state algorithm is as follows:

The bottom machine, machine 0:

if $(S + 1) \bmod 3 = R$ then $S = (S - 1) \bmod 3$.

The top machine, machine n-1:

if $L = R$ and $(L + 1) \bmod 3 \neq S$ then $S = (L + 1) \bmod 3$.

Other machines:

if $(S + 1) \bmod 3 = L$ then $S = L$,

if $(S + 1) \bmod 3 = R$ then $S = R$.

The details of how the self stabilizing systems work are, however, not important so far as restructuring the algorithms to mobile environment is concerned. Instead we need to look at the communication that occurs between any two machines.

In both the algorithms cited above and other self-stabilization algorithms, the essential communication perspective is to access the registers for the left and right neighbours. This class of algorithms are very similar to the previous class of algorithms. The presence of one or more exception machines really does not make much of a difference to the communication costs involved. Since only the neighbour's values are needed, the overhead of wireless communication would be even less, as most *neighbouring* hosts are expected to be under the same MSS except for the two at the edges.

2.4 Fixed Coordinator Based System

Distributed algorithms which involve a coordinator for running the algorithm properly, induce extra communication overhead for the coordinator. The first class of such algorithms the coordinator is the fixed coordinator set of algorithms.

Each member of this class assigns one particular host to act as the coordinator. Thus apart from the normal optimization, the communication pattern for the coordinator has to be specifically optimized.

An example of the fixed coordinator system is encountered in the case of total order atomic broadcast algorithm. Figure 1 illustrates this algorithm with a timeline diagram. The coordinator is always fixed to a particular host, e.g., host 3 in

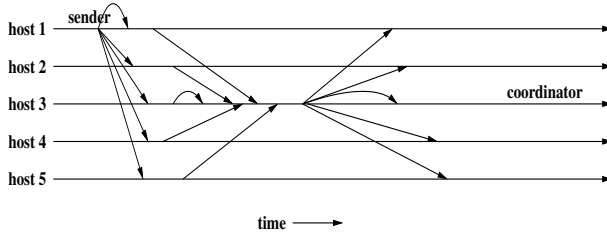


Fig. 1. Total Order Atomic Broadcast using fixed/moving coordinator.

figure 1. To modify such algorithms for execution in a mobile environment, we give special attention to the coordinator. If the algorithm is directly executed on the mobile hosts without any change, the cost incurred for each broadcast will consists of

1. the cost of broadcast: $(N - 1) * (C_{search} + C_{wireless})$,
2. the cost of sending to the coordinator: $(N - 1) * (C_{search} + C_{wireless})$,
3. the cost of broadcasting the time stamped messages back:

$$(N - 1) * (C_{search} + C_{wireless})$$

leading to a overall cost of $3(N - 1) * (C_{search} + C_{wireless})$.

This can be improved marginally if the location of the coordinator cached by each MSS. The cost in that case would be

$$2(N - 1) * (C_{search} + C_{wireless}) + (N - 1) * (C_{fixed} + C_{wireless}).$$

However, if the coordinator is placed on an MSS, the cost gets revised as follows:

1. the cost of broadcast: $C_{wireless} + (M - 1) * C_{fixed}$,
2. the cost of sending to coordinator: $(M - 1) * C_{fixed}$,
3. the cost of broadcasting back the final timestamped message:

$$(M - 1) * C_{fixed} + (N - 1) * C_{wireless}.$$

Thus, making it a total cost of $N * C_{wireless} + 3(M - 1) * C_{fixed}$.

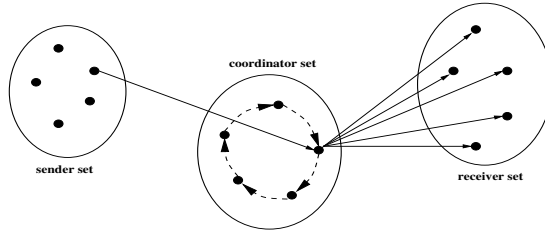


Fig. 2. Conceptual model of a moving coordinator system, the sender, coordinator and receiver sets are the same, that is comprise of the same hosts, but are shown distinctly for the sake of clarity.

2.5 Moving Coordinator Based System

The difference of a moving coordinator system from the afore-mentioned fixed coordinator system is that the role of coordinator can be performed by different mobile hosts at different times. Thus the coordinator is determined by a function of time as shown in figure 2. If the algorithm is executed on mobile hosts without restructuring, the communication cost is $3(N - 1) * (C_{search} + C_{wireless})$. We observe that as soon as a mobile host becomes a coordinator the number of accesses to it rises dramatically in a short space of time. Hence, let us modify the system as follows:

- The MH informs its MSS about change of status to coordinator, which in turn broadcasts this information to all MSSs.
- The coordinator MH uses the inform strategy during its tenure as coordinator, while other hosts use the search strategy.

With the suggested modifications,

1. the cost of broadcast: $C_{wireless} + (M - 1) * C_{fixed}$,
2. the cost of sending to coordinator: $(M - 1) * C_{fixed} + C_{wireless}$,
3. cost of broadcasting back the time stamped message:

$$(M - 1) * C_{fixed} + (N - 1) * C_{wireless},$$

4. and the additional overhead associated with change of coordinator:

$$C_{wireless} + (M - 1) * C_{fixed}.$$

Thus the total cost becomes:

$$(N + 1) * C_{wireless} + 3(M - 1)C_{fixed} + \alpha(C_{wireless} + (M - 1) * C_{fixed}),$$

assuming a change of coordinator occurs after every α broadcasts.

Comparing this with the non-restructured cost the savings are significant.

3 Synchronization

In general, whenever an operation is executed, it holds up the subsequent execution of a subset of operations. It may not be true that this subset contains only or all operations of the same type. The operations may be held up due to non-availability of the parameters to be passed to those operations. The resulting situation can be visualized as acquiring an internal lock or mutex before an operation could be executed. There may be a set of operations which would be trying to acquire the same mutex before proceeding. Hence, exactly one of these operations can proceed at a time. Thus, we may associate a subset of operations to an internal mutex in the algorithm. The concept of internal mutex is introduced only to motivate the formalization, but not included in the formalization itself.

3.1 The State of Synchronization

Generally, a state is associated with anything which has variables and the variables take different values to realize different states of the system. Our interest here is in those states of the object which concerns with synchronization. The subsequent mutexes which would be held depends on operation to be performed and the current state of the object. Also, the current state of the object determines what operations can be performed on the object. The the object itself may have a large number of states, we would like to view them only as the states concerned with the synchronization.

Definition 1. *The state of synchronization is a set of all possible sets of synchronization for that particular state of the object. Given an integer i , let y_i denote the i th state of synchronization. The number of y_i s is equal to the number of synchronization states.*

3.2 Operation Frequency Distribution

To evaluate the *goodness* or *performance* of an abstraction an object frequency distribution has to be introduced. We apply our abstractions over this frequency and calculate the number of operations performed per unit time. This number provides an idea about the *load* on the system.

3.3 The Mathematical Modeling

We have an object and concerned with every instance of an object we have an expanded abstraction set. Let us call this set A , i.e.,

$$A = \{x \mid x \text{ is an operation on an instance of the object}\}$$

Now assume that this object has a set of synchronization states. Let this set denoted by Y , i.e., $Y = \{y_1, y_2, \dots, y_n\}$. Note that y_i itself is composed of many

sets of synchronizations. For an integer i let S_i denote sets of synchronizations. Thus, $y_i = \{S_1, S_2, \dots, S_m\}$ where $S_j \subseteq A$, for $1 \leq j \leq m$. It should be noted that i denotes a state of synchronization.

If an operation $o1$ is required to be performed on an object then another operation $o2$ can not be performed concurrently on the same object if $o1$ and $o2$ are in the same set $S_i(\in y)$, where y is the set of synchronization states for the current state of the object.

Let us move to the operation frequency distribution. We have a function $f(o, y)$ defined for $o \in A$ and for $y \in Y$. We assume further that the function is normalized, i.e., $\Sigma f(o, y) = 1$, where summation runs over all $o \in A$, and for all $y \in Y$. This helps in visualizing the contribution of an operation at a particular state of the object with respect the whole. It may be noticed that occasionally there would be a few operations which cannot be performed when the object is in a particular state. This is captured by setting $f(o, y) = 0$. An operation on the object may or may not change the state of the object.

Applying the frequency distribution we can arrive at computation of concurrency loads. We compute

$$L_{ij} = \Sigma f(o, y), \forall o \in S_j, \text{ where } S_j \in y_i$$

For obvious reasons, $\Sigma_{i,j} L_{ij} \neq 1$. This is because S_i 's overlap, thus, the contribution ($f(o, y)$) can be found in multiple L_{ij} 's. In any case, we normalize this again. Thus,

$$L'_{ij} = \frac{L_{ij}}{\Sigma L_{ij}}$$

L'_{ij} represents the sum of the frequency of operations for a set of operations that try to acquire the same mutex. Thus, higher this number means that more operations of this set ($S_j \in y_i$) would be queued together and would have to wait. Also, the maximum of all L_{ij} s would represent a measure of the synchronizing bottleneck for the whole application. As the steps of an application proceeds sequentially at the MH the largest L_{ij} would be the bottleneck for the entire object access. Let us denote this maximum value by CL and call this *concurrency load*. Thus, $CL = \max(L'_{ij})$. The advantage of using the normalized L'_{ij} is that it provides a common scale to compare two different abstractions. Each of the abstractions would give a (normalized) concurrency load which can be compared to find which of the two abstraction is better.

Thus the second principle is as follows.

Principle 2 *While making an abstraction for an object try to minimize the concurrency load of the expanded abstraction set.*

It should be noted that the above principle would also improve performance in non mobile environments but the effect would be significantly large in a mobile environment because of the large number of threads that concurrently execute supplemented with the inherent latency of each operation because of poor network resources.

4 Examples

Let us illustrate the proposed abstraction on two simple examples. The first one is concerning the synchronization requirements for updating of cells of a spreadsheet. The second deals with read write lock.

4.1 Spreadsheet

Suppose a number of mobile users constantly update a spreadsheet. Let us assume two abstractions for the spreadsheet object. One, a mutex on the entire spreadsheet and another, a mutex over each cell that one wants to edit. Its easy to see that the latter case would be better in high concurrency because more than one user can simultaneously update the spreadsheet if they do not update the same cell. In the former case only one person/process can update the spreadsheet at a time.

Let us examine how two abstractions are captured through our model. In the first instance, there is one set S_1 containing all operations that can be performed on the object. Thus, $L_{11} = 1$ and $CL = 1$. In the second instance, assuming p cells are there in the spreadsheet, there are p internal mutexes, and hence 2^p states of synchronization. As there is no inter-dependence between states, actually the values of S_i s in many states would be equivalent. For each state of synchronization we have one S_1 in its set. Thus assuming that $f(o, y)$ is constant for all o and y , we have L_{ij} the same for all i and j . In fact $j = 1$ is the only value. As no S s overlap, we can safely assume that $\Sigma L_{ij} = 1$ giving $CL = \max(L_{ij}) = \frac{1}{2^p}$. Thus gives a straight forward indication that the second abstraction is better.

4.2 Read-Write Lock

Consider an object that supports read and write operations through a read-write lock. The read write lock allows concurrent reads to be performed. But it allows only exclusive writes and while a read is going on, a write cannot take place.

Thus there are two states of synchronization, the read state and the write state. Let us denote the set of synchronization of these by y_r and y_w . There are a number of read operations (many parameters) and similarly many write operations. Assume that there are p entities which can written or read. Let the expanded abstraction set be represented by

$$A = \{or_1, ow_1, or_2, ow_2, or_3, ow_3, \dots, or_p, ow_p\}$$

where or_i is a read operation and ow_i a write operation.

In state y_r , concurrent reads are supported, whereas in y_w every write operation is exclusive, Thus,

$$\begin{aligned} y_w &= \{\underbrace{ow_1, ow_2, ow_3, \dots, ow_p}_{}, or_1, or_2, or_3, \dots, or_p\} \\ y_r &= \{\underbrace{or_1}_{}, \underbrace{or_2}_{}, \underbrace{or_3}_{}, \dots, \underbrace{or_p}_{}, \underbrace{ow_1, ow_2, ow_3, \dots, ow_p}_{}\} \end{aligned}$$

The sets of synchronization, are indicated by underbraces. We do not worry about those elements which are not grouped together as these operations cannot be performed in that state of the object. Consequently, in the operation frequency distribution

$$f(ow_i, y_r) = 0 \text{ and } f(or_i, y_w) = 0$$

Hence, the evaluations for these operations would any way cancel out. Assuming $f(o, y)$ to be constant for all o and y , and from the fact that $\Sigma f(o, y) = 1$ we deduce that,

$$f(o, y) = \frac{1}{2p}$$

Now, let us evaluate L_{ij} . Note that $L_{w_i} = \Sigma f(ow_i, y_w)$ for $1 \leq i \leq p$. Similarly, $L_{r_j} = f(or_j, y_r)$ for $1 \leq j \leq p$. Thus,

$$L_{w_i} = p \frac{1}{2p} = \frac{1}{2}, \text{ and } L_{r_j} = \frac{1}{2p}$$

Incidentally, no sets of synchronization overlaps. Thus $\Sigma L_{ij} = 1$, and no normalization is required. Thus, the concurrency load $CL = \max\{L_{ij}\} = \frac{1}{2}$. This implies a read write lock is better than a single mutex based lock (in which case $CL = 1$). Note that in typical scenarios, the value of $f(or, y_r)$ is much larger than $f(ow, y_w)$ simply because the number of read operations is much more than the number of write operations. In that case it can be easily seen that the above analysis would give a better value for CL .

References

1. BADRINATH, B. R., ACHARYA, A., AND IMIELINSKI, T. Impact of mobility on distributed computations. *Operating Systems Review* 27 (1993), pp. 15–20.
2. BADRINATH, B. R., ACHARYA, A., AND IMIELINSKI, T. Structuring distributed algorithms for mobile hosts. In *Proceedings of ICDCS* (1994), pp. 21–28.
3. DIJKSTRA, E. W. Self stabilizing system in spite of distributed control. *CACM* 17 (1974), pp. 643–644.
4. GHOSH, R. K., CHAKRABORTY, A., AND KUMAR, R. Restructuring distributed algorithms for mobile computing environment. Tech. rep., Department of CSE, IIT-Kanpur, April 2002.
5. GRANT, W. C. Wireless coyote: A computer supported field trip. *CACM* 36 (1993), pp. 57–59.
6. LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *CACM* 17 (1974), pp. 453–455.