Scheduling Concurrent RPCs in the Globe Location Service

Gerco Ballintijn, Magnus Sandberg, Maarten van Steen

Vrije Universiteit, Department of Mathematics and Computer Science,
De Boelelaan 1081a, 1081 HV, Amsterdam, The Netherlands
{gerco,steen}@cs.vu.nl

Keywords: concurrent RPC, distributed objects, wide-area computing

Abstract

Globe is a wide-area distributed system in which an object can be located through its location-independent identifier. This is done by means of a worldwide location service. In contrast to comparable services, the approach that is followed in Globe allows objects to be highly mobile, replicated, or physically distributed. In addition, our algorithms adapt dynamically to an object's behavior, resulting in an efficient and above all, scalable approach.

The algorithms for updating and looking up an object's location are expressed as high-level operations on a worldwide search tree. We have designed and implemented a middleware layer providing all the necessary network communication. In this paper, we show that such a layer hardly introduces any additional overhead. The important consequence is that our location service can be designed and implemented at a high level of abstraction. Compared to the design and implementation of comparable worldwide services, this approach is quite unique.

1 The Globe Location Service

1.1 Background

In Globe, distributed objects provide services to processes. To contact an object, a process first retrieves its *object handle* from a name server or by other means. An object handle uniquely identifies an object in Globe. The process then asks the Globe location service to provide it with the object's *contact addresses*. Every object in Globe has one or more contact addresses. These addresses specify how and where a process can contact an object. For example, the contact address of a WWW server object can specify that a process should use the HTTP protocol at address 130.37.24.11 port 80.

The reason for separating object handles and contact addresses is that they have different characteristics and uses. These differences become clear when we look at migrating and replicated objects. When an object migrates, its object handle does not change as it still refers to the same object. However, contact addresses do change when an object migrates, thus reflecting the object's new location. Likewise, when an object is replicated, its object handle is used to refer to all active copies, since they are all functionally the same. Each replica, however, has its own contact address.

By separating object handles and contact addresses, we are able to implement migrating and replicated objects. It also allows us to support physically distributed objects [3]. We do, however, need a way to find the contact addresses of an object. The location service provides this service. It maps an object handle to one or more contact addresses.

The Globe location service provides three basic operations: insert, delete and lookup. The insert operation informs the location service that an object can be contacted at a (new) contact address. The delete operation informs it that an object can no longer be contacted at a certain contact address. The lookup operation searches for contact addresses of a designated object. The location service provides only a mapping from object handle to contact address, not vice versa. Given the fact that the Globe location service works on a global scale, it should be able to handle large numbers of update and lookup requests, and deal gracefully with network partitions and long network delays.

In this paper we are mainly interested in the speed of propagation of modifications in the location service. For example, how long does it take for a contact address to be visible through out the system after it has been inserted?

1.2 The logical structure

The Globe location service partitions a worldwide network in disjunct *basic regions*. A contact address is always situated into exactly one basic region. Regions are recursively combined into larger regions, ending in one region covering the whole network, as shown in Fig. 1. Every region has an associated *directory node*, which stores information about the object handles in its region. These nodes together form a distributed search tree, which represents the hierarchical partitioning of the network. The directory nodes of the basic regions are the leaf nodes of the tree. The tree structure is internal to the location service and not visible to its users. A basic region is expected to have the size of a departmental LAN.

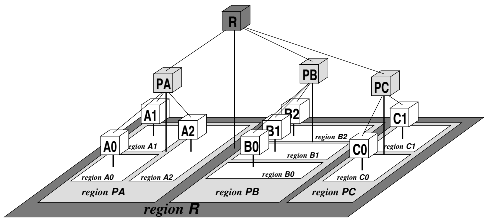

Figure 1: Hierarchical region partitioning

A directory node associates a *contact record* with every object handle it knows. A contact record stores information about the object handle. In the normal case, leaf nodes store contact addresses and higher level nodes store *forwarding pointers* to a leaf node. A forwarding pointer indicates that the contact address may be found in the child's region (subtree). By following a path of forwarding pointers from the root directory node, one can find the directory node where a contact address is stored.

When an object migrates between basic regions, contact addresses need to move from one leaf node to another, and consequently forwarding pointers need to be updated. When an object migrates often, this can lead to a lot of work. However, when the contact addresses are stored at a higher level in the tree (representing the larger region in which the migration takes place), the placement of the contact addresses can remain the same, thus preventing updates on the forwarding pointers. The contact addresses themselves still need to be updated.

When a process wants to know an object's contact addresses, it starts a lookup operation at the leaf node of its basic region. This request is propagated up the tree, searching in ever increasing regions, until a nonempty contact record is found, i.e. a record that contains either a contact address or a forwarding pointer. Then, a path of forwarding pointers starting in this record is followed until a contact record with contact addresses is found. The design of an efficient lookup algorithm is currently an important subject of our research. It will not be further discussed in this paper. Likewise, we omit any discussion on scalability and optimizations. The interested reader is referred to [4] for further details.

1.3 Update Operations

The insert and delete operations are referred to as update operations. They have two parameters, an object handle and a contact address. The contact address is to be inserted in or deleted from the object handle's contact record.

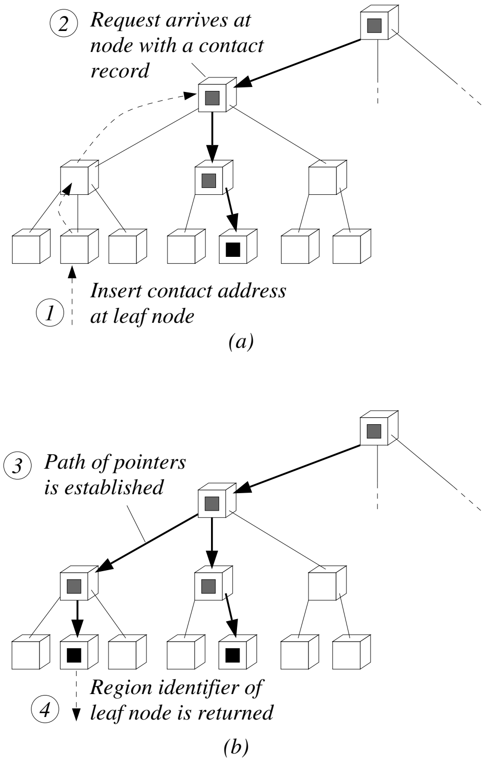

Figure 2: The insert operation

An insert operation starts at the leaf node of the basic region to which the contact address belongs. In the normal case the contact address is stored in the leaf node and a path of forwarding pointers is built by recursively inserting a forwarding pointer at every node starting at the leaf. The insert operation is completed as soon as a node is reached that already had a forwarding pointer, or otherwise at the root. This is illustrated in Fig. 2.

A delete operation starts at the leaf node of the region where the contact address was inserted. Normally, a delete operation deletes the contact address at a leaf node. If the leaf node no longer contains contact addresses or forwarding pointers, the path of forwarding pointers to this node is recursively removed.

The communication pattern of each update operation is basically the same. It first reads the contact record to decide whether it should update the record or not. The operation then generally forwards the update request to the parent and is suspended until the parent responds. Depending on the parent's response, the contact record is then either updated or left in its original state. This can be expressed as shown in Fig. 3.

```
operation update(oid : OID, addr : Address) is
  if perform update here then
    — First check with parent if update is allowed. As a side effect,
    — the parent possibly also updates its own local contact record.
    let response := call update(oid, addr) at parent;
    if response = OK then
      — Parent agrees with update at this node
      perform update;
      if caller is allowed to continue update
        then return OK;
        else return DONE;
      fi
    else
      — Update was completed at a higher level. This level
      — and lower ones are not allowed to perform update locally.
      return DONE;
    fi
  else
    — Simply forward the update request to the parent.
    call update(oid, addr) at parent;
    return DONE
  fi
endupdate
```

Figure 3: Outline of a general update operation.

It is important that update operations do not violate the consistency of the tree. Consistency is formulated on a per-object basis. In particular, we say that the tree is **globally consistent** for a specific object O, if it conforms to the following two predicates:

Pointer The contact record for O at a node N stores a forwarding pointer to a child node of N if and only if the contact record for O at that child node is nonempty.

Exclusion A contact record for O at a node N contains a forwarding pointer to child C only if it does not contain a contact address for the region represented by C.

These predicates imply that a contact record at a leaf node can never contain a forwarding pointer. Also, if a contact record contains an address for object O on behalf of its child C, then all directory nodes in the subtree rooted at C will have empty contact records for O.

To guarantee exclusive access to a contact record, a node would normally have to deny successive invocations until the current operation has completed. Effectively, this means that an update request issued at a leaf node cannot be handled before the previous one has been processed at every node in that request's invocation chain, possibly up to the root of the tree. For a wide-area system, this strict sequential behavior is unacceptable. Instead, what we need is a mechanism that will allow us to schedule and execute an operation before the previous one has completed, but that would lead to the same results as with strict sequential invocations. In particular, we want to continue with the next operation as soon as the current one is waiting for a response from the parent. A successive operation should thus be able to base its decisions on *tentative data* in such a way that the tree eventually becomes globally consistent. Operations on tentative data that lead to eventual global consistency is accomplished through **view series**.

1.4 Views and View Series

A **view** on a variable v is an expression formulated in terms of v that can be evaluated, but which leaves the original value of v unaffected. A view on a variable v can be evaluated only when it has been appended to a **view series** associated with v. The value of a view series of v results from evaluating the views in that series in the order that they were appended, leading to a *tentative value* of v. To illustrate, consider the following example (we use the notation "$v \oplus S$" to denote that the series S is associated to v).

```
let x : Integer = 4;
let vseries : view series of x;
append view {self + 1} to vseries;
append view {self * 2} to vseries;
— vseries = 4 ⊕ {x + 1, 2 · x}, with value 10
```

We declare an integer variable x and associated view series vseries. The pseudo-variable **self** points to the variable for which the corresponding view series is defined, in this case x. In the example, the viewed value of vseries is always $2 * (x+1)$ regardless how x is changed. The value of a view series is obtained by taking the actual value of the associated variable and evaluating all view expressions.

The view at the head of a view series, i.e. the least recently appended one, can be **applied** by evaluating its expression and changing the value of the associated variable accordingly. The view is then removed from the view series. A view can also be directly **removed**, i.e. without applying it.

To properly serialize concurrent updates, we associate a view series with each contact record. Each operation first appends a view corresponding to the update it wants to make, so that the contact record is left in a tentative state, as if the update were completed. When the parent responds, the operation continues by either applying the appended view (i.e., when the update at the current node succeeded so that the view can be turned into authoritative data), or removing the view (in the case the parent did not permit the update). While waiting for the parent to respond, the next operation is scheduled, which in turn is executed according to the tentative state of the contact record. This leads to the adaptation shown in Fig. 4.

We demand that operations are invoked at the parent in the same order as called by a child. Operations called by different children may be invoked in an arbitrary order. Furthermore, an invoked operation is assumed to be executed exclusively and nonpreemptively until suspended on a **call** primitive, or until the

```
operation update(oid : OID, addr : Address) is
  if perform update here then
    append view update to contact record;
    let response := call update(oid, addr) at parent;
    if response = OK then
      apply view to contact record;
      if caller is allowed to continue update
        then return OK;
        else return DONE;
      fi
    else
      remove view from contact record;
      return DONE;
    fi
  else
    call update(oid, addr) at parent;
    return DONE
  fi
endupdate
```

Figure 4: Outline of a general update operation using view series.

operation completes. The execution at the caller's side continues in the order of the completion of the **call**ed operation at the parent. How these semantics are implemented is described next. Details on our update algorithms can be found in [2].

2 RPC Design Issues

Our sequential communication model suggests the use of RPCs [1]. Note that in our algorithms this leads to series of chained RPCs from leaf to root node. A leaf node has to wait until the chained RPCs are completed before it can finish. Race conditions can occur when concurrent invocations try to modify the same local contact record. This can be avoided by making update operations atomic.

As we have pointed out, the combination of mutual exclusion and chained RPCs presents a problem: invoking an RPC while retaining exclusive access to a contact record leads to the record being inaccessible until the nodes in the chained RPCs finish their operation. We would thus like to have concurrent, but serialized RPCs. To this end, we have designed and implemented a separate scheduling layer that provides concurrent RPCs, but which serializes calls to other nodes. Clearly, this layer introduces an additional overhead. We are interested whether this overhead is justified by the intended gains, namely a controlled concurrency of RPCs allowing us to keep our update operations in the location service relatively simple.

The rest of this section describes and analyzes the sequential and concurrent RPC system, and ends with a comparison between the two.

2.1 RPC Analysis

In the analysis we are interested in the propagation speed of modifications. We want to know how long it takes for m update requests to finish. To provide us with an upper bound, we use a worst case scenario where a contact record at the root of the tree needs to be modified, resulting in a call chain from leaf to root. This scenario provides the most work and the most communication overhead.

To simplify matters, we focus on providing only an upper bound. We show that we can establish significant improvements by using concurrent RPCs. A more detailed analysis is deferred to a forthcoming paper.

In the analysis the following variables are used.

t_{delay} time spent from issuing first request to receiving reply from the last request.

t_{node} time spent in a node doing the work for a request.

t_{link} delay between a child node sending a request message and message reception at its parent, combined with the delay between the parent node sending a reply message and message reception at the child node (time spent on the 'wire').

For simplicity we assume that the network delays are constant as well as the time it takes to do the work at a node. We also assume that the underlying communication layer provides reliable communication.

The tree is of height n and there are m requests issued.

2.2 Sequential RPC

A request issued at a leaf node, leads in our worst case scenario to a chain of RPCs that have to cross $n-1$ links. This gives us a total latency of $n * t_{node} + (n-1) * t_{link}$. Since a new request is scheduled when the preceding one finishes, this delay is also the delay for the next request to be scheduled. Therefore the total delay is:

$$t_{delay} = m * (n * t_{node} + (n-1) * t_{link})$$

2.3 Concurrent RPC

To introduce concurrency, we make a distinction in the location service algorithms between *tentative* and *authoritative* data. Tentative data is data created by an operation for which no reply message has yet been received, i.e. the call is still waiting for a reply from the parent. Authoritative data has been fully acknowledged by the parent.

In this model a new request is scheduled as soon as the current request blocks on an RPC. This allows a new request to be started on this node, while the current request is started at the parent.

The concurrent RPC system guarantees the following:

- Request messages are handled in the order they are sent.

- Reply messages are handled in the order that their associated request was sent.

This implies that a series of blocked RPCs return in the same order they blocked. This way serialized RPC semantics can be maintained.

In the concurrent RPC system, RPC requests are handled independently, except that ordering is maintained. If we furthermore ignore that requests and replies are interleaved at a single node, the last request returns after:

$$t_{delay} = (m-1) * \frac{1}{2} * t_{node} + n * t_{node} + (n-1) * t_{link}$$

The term $\frac{1}{2} * t_{node}$ reflects the delay between two consecutive requests issued at a leaf node. The last two terms correspond to the total latency as the result of propagating a request to the root of the tree, and returning an acknowledgment.

2.4 Comparison

By comparing the two delays, we can see that in the sequential case every new RPC adds $n * t_{node} + (n-1) * t_{link}$ units to the delay, and in the concurrent case $\frac{1}{2} * t_{node}$. Considering that in a global network $t_{node} \ll t_{link}$, we have gained a lot with this scheduling. This assumption should, however, be validated by measurements.

A big problem with sequential RPCs is that pending requests cannot be handled during network partitions, not even in the connected subtree. This happens because a leaf node has to receive a reply message for its current call before it can send a new request message. The system thus has to wait until the network is reconnected before it can continue. This is not an acceptable situation in a distributed system. Concurrent RPCs do not have this problem. The system keeps on processing requests, since a new operation is started as soon as the current does an RPC.

Note that the effect of an update operation is immediately visible as tentative data in the subtree rooted at the node where the request is being processed.

3 RPC Implementation

In the current version of the Globe location service, every directory node in the tree is a separately running program. A directory node knows the contact addresses of its parent and child nodes by means of configuration files.

A directory node consists of three layers as depicted in Fig. 5. The upper layer is the application layer. It implements the update and lookup algorithms and their associated data structures. The middle layer is the scheduling layer, which is responsible for the RPC scheduling. It is fully described in the rest of this section. The lower layer is the communication layer which provides a convenient communication interface for reliable message streams. Its implementation offers an asynchronous mode of communication, with a nonblocking message send primitive and a callback for notification of message arrival.

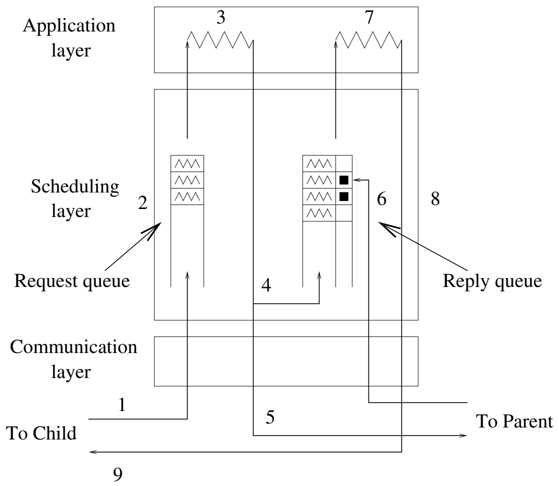

Figure 5: Flow of control during an RPC

The scheduling layer schedules the threads that perform the operations in the location service. It implements the scheduling policy required for concurrent RPCs. It associates two thread queues with every contact record: the *request queue*, which queues threads that are ready to handle an incoming RPC request, and the *reply queue*, which queues threads that are blocked on an RPC. The scheduling layer and its use of the two queues can best be explained by following what happens when an RPC request is handled. (see Fig. 5)

1. When a request message arrives at a node, the communication layer calls a callback function to notify the scheduling layer. The scheduling layer then creates a thread to handle the request.

2. The thread first appends itself to the request queue and blocks. It remains blocked until it is at the head of the queue and no other thread has access to the local contact record.

3. When the thread unblocks, it starts running the location service code in the application layer. It continues until it needs to perform an RPC to the parent node.

4. At that point, the scheduling layer blocks the thread and adds it to the reply queue. Since the thread is now blocked, new threads can be scheduled to access the contact record.

5. The scheduling layer calls the communication layer to asynchronously send the RPC request message to the parent.

6. As soon as the parent replies, the thread, which is still blocked in the reply queue, is marked as runnable. However, the thread remains blocked until it is at the head of the reply queue. At that point it can be rescheduled provided that no other threads have access to the local contact record.

7. The thread then continues running the code in the application layer.

8. When the request is finished, the scheduling layer saves the return value of the routine which handled the request and destroys the thread.

9. It then calls the communication layer to asynchronously send the reply message containing the return value of the routine to the child node.

4 Measurements

For our measurements, we used a tree with four nodes, as shown in Fig. 6.

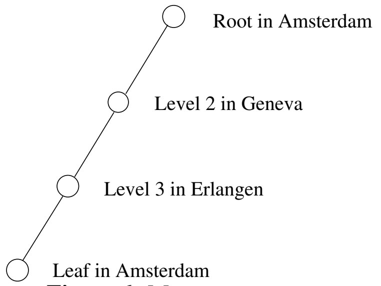

Figure 6: Measurement setup

To use the communication delays of a real distributed location service, the measurement setup comprised three geographically distant sites. We used sites in Erlangen, Geneva and Amsterdam. The Erlangen and Geneva sites each had one node, while the Amsterdam site had two.

The total delay time was measured using different number of requests. The current implementation can only handle up to approximately 130 concurrent request due to operating system constraints.

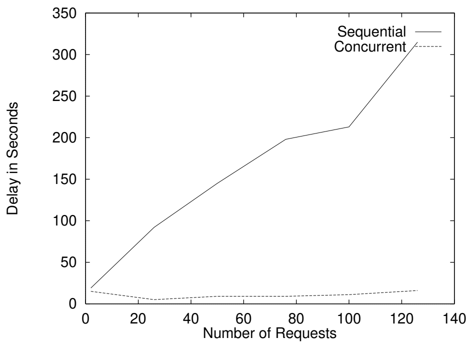

Figure 7: Measurement results

As can be seen in Fig. 7, the total delay time increases linearly with sequential RPCs. This is to be expected. The total delay time of concurrent RPCs seems to rise slightly, but the increase is much less than that of sequential RPCs. These results provide evidence that our three-layer approach is feasible.

5 Conclusions

In this paper, we have described the current state of our research on the Globe location service. We have developed an intermediate layer that allows us to make use of concurrent, but serialized RPCs within a node. The benefit of this approach is that our algorithms for updating and looking up contact addresses can be expressed as high-level operations on a worldwide search tree. The drawback is that we introduce additional overhead. A simplified analysis, backed up by a first batch of experiments, indicate that the extra overhead is almost negligible in the face of wide-area communication.

Building the prototype has also revealed some implementation problems. At present, the size of our experiments is limited due to operating system constraints. In particular, the number of concurrent threads that can be simultaneously scheduled within a single process is limited. A redesign of our scheduling layer is therefore necessary to conduct further experiments.

References

[1] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comp. Syst.*, 2(1):39–59, Feb. 1984.

[2] F.J. Hauck, M. van Steen, and A.S. Tanenbaum. Algorithmic Design of the Globe Location Service. Technical report no. IR-413, Vrije Universiteit, Amsterdam, Dec. 1996.

[3] P. Homburg, M. van Steen, and A.S. Tanenbaum. An Architecture for a Wide Area Distributed System. In *Proc. Seventh SIGOPS European Workshop*, pp. 75-82, Connemara, Ireland, Sep. 1996. ACM.

[4] M. van Steen, F.J. Hauck, and A.S. Tanenbaum. A Model for Worldwide Tracking of Distributed Objects. In *Proc. TINA'96*, pp. 203-212, Heidelberg, Germany, Sep. 1996. Eurescom.