Determining Consistent States of Distributed Objects Participating in a Remote Method Call

Magdalena Sławińska and Bogdan Wiszniewski

Faculty of Electronics, Telecommunications and Informatics
Gdańsk University of Technology
Narutowicza 11/12, 80-952 Gdańsk, Poland
`{magg, bowisz}@eti.pg.gda.pl`

Abstract. The article presents the first component of a new approach for testing distributed object-oriented applications called *TestByRep* which is based on the concept of replication of object states. The paper describes key ideas in *TestByRep* like: *hash clocks* for ordering events in distributed object-oriented systems with the unknown number of objects, the *E-path* for representing the execution of the method, construction of the *E-tree* for determining states of objects involved in a remote method call and introduces the *recovery condition* in order to assure that the determined state is consistent.

1 Introduction

Testing non-distributed applications is easier than testing distributed ones since in non-distributed applications it is usually possible to use the *cyclic debugging* technique [1, 2]. The cyclic debugging technique is based on the approach of repeatable execution of a program in order to exercise its code and localise errors. Although it is a very simple technique, it is commonly used by testers and programmers of non-distributed applications. However, adapting the cyclic debugging technique to distributed applications is not straightforward and requires special mechanisms which must be incorporated into the distributed application, the execution platform or the operating system. It is because in distributed systems testers meet difficulties like: determining a global consistent state of a distributed application, physical distribution of application's components, accessibility of application's components, the lack of global clock which makes impossible to reckon about the order of the events in a distributed application [3].

This paper describes the first component of a new methodology called *TestByRep* based on the concept of replication of object states in order to enable the cyclic debugging technique for distributed object-oriented applications (DOA). The idea of replication is widely used in order to increase reliability and accessibility as well as to provide fault-tolerance and support load-balancing [4, 5, 6]. The methodology *TestByRep* shows that the idea of replication can be also applied to test and debug DOA.

TestByRep consists of three main components: (1) the model and key concepts, (2) the infrastructure to capture the relations among objects and object states during the execution of DOA and (3) the *method driven recovery approach* which comprises *reexecution*, *simulated execution* and *selective execution* of replicated objects [7].

The description of the infrastructure to capture the relations among objects in DOA can be found in [8]. In this paper we present fundamental concepts of the methodology *TestByRep*: *hash clocks*, *E-paths* and *E-trees*.

2 Hash Clocks

To order the events in DOA, we introduce the concept of *hash clocks*. The *logical clock* and two important conditions: (1) the *clock condition* and (2) the *strong clock condition* were defined in [3]. The logical clock (it is a function) assigns numbers to events in the system. In order to work properly it should satisfy the clock condition. Events in the system have *timestamps* which are concrete values of the logical clock. Timestamps are assigned to events in the system. The clock condition says that earlier events in the system should have lower timestamps [3]. And when it is possible to draw a conclusion based on timestamps about the order of the events in the system, we say that the logical clock satisfies the strong clock condition [3]. Logical clocks can be defined as numbers [3], vectors with the *a priori* known and constant size [9] or matrices also with the *a priori* given constant size [9, 10, 11]. However, sometimes in DOA it is difficult to predict the number of objects in the application, especially if *foreign* objects are considered [8]. A distributed object-oriented application consists of cooperating objects scattered over the network. A tester can have an access to the source code of the object and such an object is called *native* object or s(he) cannot have an access to the source code and such objects are called *foreign* objects [12], e.g., services which are used by native objects.

In this context the assumption about the constant and known sizes of vectors or matrices is irrelevant. In [13] were proposed logical clocks which take into account the dynamic and unknown number of processes in the system. However, they require an additional process which assigns subsequent numbers to new processes or before registering a new process in the system, all other processes must agree on the number which should be assigned to this new process. The presented approach can be appropriate for non-object distributed systems like PVM or MPI (in fact it was proposed for PVM applications), however DOA requires a new approach.

In this paper we propose a new logical clock called a *hash clock*. The construction of the hash clock is based on the *hash table*. The hash table is the set of key-value pairs with effective search of elements [14]. We assume that elements can be dynamically added to the hash table. We also assume that objects in the distributed object-oriented system have unique identifiers. Symbol id_o means the identifier of object o and HC_o denotes the hash clock of object o. Hash clock HC_o is a set of key-value pairs where keys are object identifiers while values are natural numbers representing the logical clock of a given key-object from the point of view of object o. For example: $HC_o = \{(id_o, C_o), (id_{o_1}, C_{o_1}), \ldots, (id_{o_k}, C_{o_k})\}$, where $k = 1, 2, \ldots,$ means that from the point of view of object o: object o has the clock value C_o, object o_1 has the clock value C_{o_1}, and so on. Let the function $HC_o()$ give a value of the key argument in hash clock HC_o, i.e., $HC_o(id_{o_1}) = C_{o_1}$. Values of hash clocks will be called *hash timestamps* and denoted by symbol HTS.

Before presenting the algorithm for updating the hash clock, first we should define operations of adding the new value to the hash clock and merging a hash clock with a hash timestamp. The operation of adding the new element to the hash clock is defined as follows:

$$HC_o.add(id_{o_1}, C_1) \equiv HC_o := HC_o \cup \{(id_{o_1}, C_1)\} \tag{1}$$

We assume that if it is required operation *add*() increases the size of the hash clock and then adds the element to it. The symbol Id_{HC_o} denotes the set of identifiers of the hash clock of object o and Id_{HTS} denotes the set of identifiers of hash timestamp HTS, e.g., $Id_{HC_o} = \{id_o, id_{o_1}, \ldots, id_{o_k}\}$. Function $mergeMax()$ defines how to merge a hash clock and a hash timestamp and is described by Algorithm 1.

Algorithm 1 (Function $mergeMax(HC_o, HTS)$)

```
1. $\forall_{id \in Id_{HTS}}\ (id \notin Id_{HC_o}) \Rightarrow (HC_o.add(id, HTS(id))$.
2. $\forall_{id \in Id_{HC_o}}\ (id \in Id_{HTS}) \Rightarrow (HC_o(id) := max(HC_o(id), HTS(id)))$.
```

Function $mergeMax()$ increases hash clock HC_o (if it is required) and compares the values of logical clocks of relevant identifiers of HC_o and HTS, and then chooses the greater value and assigns it to the relevant object identifier in HC_o.

We assume that objects interact with other objects only by method calls which can be modeled as requests and replies [6, 8]. By the *internal* event we understand the event which does not require the communication with another object, e.g., the event of invoking by an object its own method. Otherwise the event is called *external*, i.e., requires the interaction with another object. In the context of requests and replies the method call can be modeled as the event of (1) sending request req by object o_1 to object o_2 denoted by $s(req)$, (2) receiving request req by object o_2 denoted by $r(req)$, (3) performing relevant actions by o_2, (4) sending reply rep by object o_2 to object o_1 denoted by $s(rep)$, and finally (5) receiving reply rep by object o_1 from object o_2 [8]. If it is not relevant to distinguish replies and requests, they will be called *messages* and denoted by symbol m. We assume that messages carry hash timestamps of their senders.

Now, the algorithm of updating the hash clock can be introduced.

Algorithm 2 (Updating the hash clock)

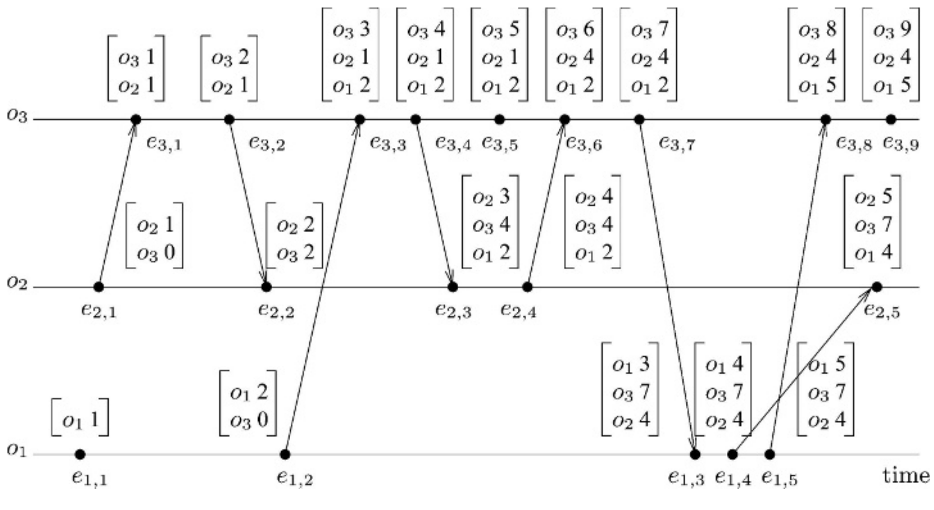
```
1. $HC_o := \{(id_o, 0)\}$.
```

```
2. For each new event e in the system:
    1). $HC_o(id_o) := HC_o(id_o) + 1$,              if event e is an internal event
    2). $HC_o := mergeMax(HC_o, HTS(m))$,   if $e = r(m)$
        $HC_o.add(id_p, 0)$,                     if $e = s(m)$ and $id_p$ is a receiver
                                                of $m$ and $id_p \notin Id_{HC_o}$
        $HC_o(id_o) := HC_o(id_o) + 1$.
```

Firstly, the hash clock is initialized and then after each internal event the clock value referring to object o is increased by 1. Then, after each event (external or internal) the hash clock is increased by 1. However, in the case of the receive event the current hash clock is compared with the received hash timestamp of the received message and updated according to Algorithm 1. In the case of the sending event if a receiver is not in the hash clock, the receiver is added to the hash clock and its clock is initialized with 0.

Figure 1 presents Algorithm 2. Let's notice that the size of the hash clock for each object settles at the number of objects in the system if objects in the system interact and information about hash clocks scatters over objects in the system. Let's consider for example the send event $e_{1,2}$ from Figure 1. The hash timestamp assigned to event $e_{1,2}$ is $HC_{o_1}(e_{1,2}) = \{(o_1, 2), (o_3, 0)\}$. Next, this hash timestamp is sent to object o_3. Since HC_{o_3} does not contain the identifier of object o_1, pair $(o_1, 2)$ is added to HC_{o_3} according to Algorithm 1, and the rest elements are updated according to Algorithm 1. Finally, this results in $HC_{o_3}(e_{3,3}) = \{(o_3, 3), (o_2, 1), (o_1, 2)\}$.

Since hash clocks can have different sizes in order to compare them we need to normalize them. Let's define operation *extend*():



o_3
o_2
o_1
time
$e_{3,1}$ $e_{3,2}$ $e_{3,3}$ $e_{3,4}$ $e_{3,5}$ $e_{3,6}$ $e_{3,7}$ $e_{3,8}$ $e_{3,9}$
$e_{2,1}$ $e_{2,2}$ $e_{2,3}$ $e_{2,4}$ $e_{2,5}$
$e_{1,1}$ $e_{1,2}$ $e_{1,3}$ $e_{1,4}$ $e_{1,5}$
$[o_3\ 1;\ o_2\ 1]$ $[o_3\ 2;\ o_2\ 1]$ $[o_3\ 3;\ o_2\ 1;\ o_1\ 2]$ $[o_3\ 4;\ o_2\ 1;\ o_1\ 2]$ $[o_3\ 5;\ o_2\ 1;\ o_1\ 2]$ $[o_3\ 6;\ o_2\ 4;\ o_1\ 2]$ $[o_3\ 7;\ o_2\ 4;\ o_1\ 2]$ $[o_3\ 8;\ o_2\ 4;\ o_1\ 5]$ $[o_3\ 9;\ o_2\ 4;\ o_1\ 5]$
$[o_2\ 1;\ o_3\ 0]$ $[o_2\ 2;\ o_3\ 2]$ $[o_2\ 3;\ o_3\ 4;\ o_1\ 2]$ $[o_2\ 4;\ o_3\ 4;\ o_1\ 2]$ $[o_2\ 5;\ o_3\ 7;\ o_1\ 4]$
$[o_1\ 1]$ $[o_1\ 2;\ o_3\ 0]$ $[o_1\ 3;\ o_3\ 7;\ o_2\ 4]$ $[o_1\ 4;\ o_3\ 7;\ o_2\ 4]$ $[o_1\ 5;\ o_3\ 7;\ o_2\ 4]$

Fig. 1. The hash clock for the dynamic unknown number of objects in the system

Definition 1 (Operation extend(HTS_1, HTS_2)) *Let HTS_1 and HTS_2 be hash timestamps. We say that HTS_{ext} is the extended hash timestamp of HTS_1 with respect to HTS_2 if HTS_{ext} was constructed according to the following rules:*

1. $HTS_{ext} := HTS_1$.
2. $\forall_{id \in Id_{HTS_2}} \quad (id \notin Id_{HTS_1}) \Rightarrow HTS_{ext}.add(id, 0)$.

Operation *extend*() returns the hash timestamp which contains all object identifiers and relevant values from its first argument and all object identifiers from its second argument with value 0 if the first argument does not contain them. Let's consider for example $HTS_1 = HTS(e_{2,2}) = \{(o_2, 2), (o_3, 2)\}$ and $HTS_2 = HTS(e_{3,4}) = \{(o_3, 4), (o_2, 1), (o_1, 2)\}$ from Figure 1. Then $HTS_{ext} = extend(HTS_1, HTS_2) = \{(o_1, 0), (o_2, 2), (o_3, 2)\}$.

Considering operation *extend*() it is possible to define operators $\leqslant, <$ and $||$ for comparison of hash timestamps [7]. Informally $HTS_1 \leqslant HTS_2$ when for all identifiers in HTS_1 their values are not greater than relevant values in HTS_2. We say that $HTS_1 < HTS_2$ if $HTS_1 \leqslant HTS_2$ and exists at least one identifier id in HTS_1 that $HTS_1(id) < HTS_2(id)$. When $HTS_1 || HTS_2$, i.e., the hash timestamps are *concurrent*, then $\neg(HTS_1 < HTS_2) \wedge \neg(HTS_2 < HTS_1)$.

We can prove that hash clocks with updating by Algorithm 2 and operation *extend*() satisfy the clock condition and strong clock condition [7]. It means that hash clocks can be used to conclude about the order of events in the system and construct consistent state of a set of objects.

3 Consistent States of Distributed Objects

The states of objects registered in corresponding logs during the execution of DOA are called *checkpoints* [10]. They will be denoted as $\times$ on diagrams and instead of $e_{i,j}$ we will use symbol $s_{i,j}$. In order to check if a given set of checkpoints is consistent it is sufficient to check if timestamps corresponding to checkpoints are concurrent. If at least one pair of the set of checkpoints is not concurrent the state of the set of objects represented by these checkpoints is not consistent [15].

In order to determine the consistent state of the set of objects we can represent a method call as an *execution path*, shortly called *E-path*.

3.1 *E-Path*

The *E-path* is the sequence of events beginning from the event of sending a request and ending with the event of receiving the reply corresponding to this request. Let's consider Figure 2.

Since $e_{1,2} = s(req_1)$ and $e_{1,3} = r(rep_1)$, they determine the following *E-path* $= \langle e_{1,2}, e_{2,2}, e_{2,4}, e_{3,2}, e_{3,3}, e_{2,5}, e_{2,7}, e_{3,5}, e_{3,7}, e_{4,4},$
$e_{4,5}, e_{3,8}, e_{3,10}, e_{5,2}, e_{5,3}, e_{3,11}, e_{3,13}, e_{2,8}, e_{2,10}, e_{1,3}\rangle$.

Having the *E-path* of the method call it is possible to determine which objects in the system are involved in the method call. For example in Figure 2 the following objects participate in calling a method in request req_1: $\{o_1, o_2, o_3, o_4, o_5\}$ since all of them belongs to *E-path*(req_1).

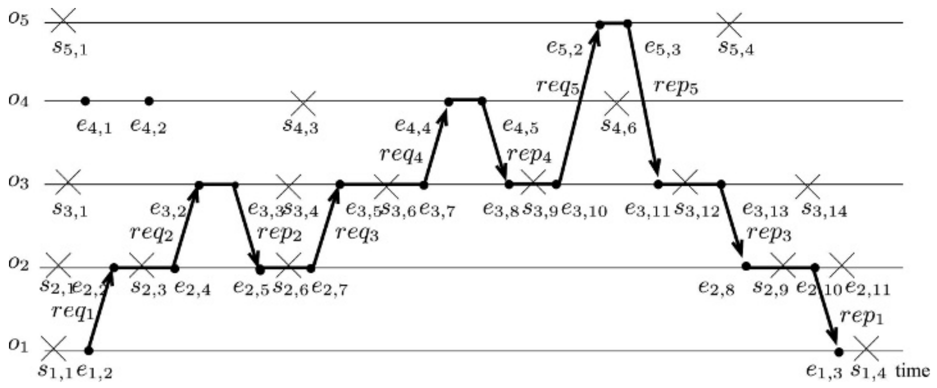


Fig. 2. The E-path

In order to determine the consistent state before the method call, the *E-tree* can be constructed.

3.2 *E-Tree*

The *E-tree* is the tree of the execution of the method. It is an ordered tree [14] where the root of the tree is the request of the method execution which is sent by a caller to a callee and nodes are requests which belong to *E-path* of the request in the root node. Earlier requests are placed in the most left nodes in the *E-tree*. The *E-tree* for request req_1 from Figure 2 is shown in Figure 3.

Each level of the *E-tree* corresponds to the so-called *method execution interval* of the direct parent in the *E-tree*. The method execution interval denoted by I_m is the period of time determined by the event of receipt of the request by an object and the event of sending the reply to the request. Informally, the method execution interval is the period of time when the callee perform the method requested by the caller. For example in Figure 2: $I_m(req_1) = \langle e_{2,2}, e_{2,10}\rangle$ and $I_m(req_3) = \langle e_{3,5}, e_{3,13}\rangle$. Considering the *E-tree* from Figure 3 we can see that $req_2, req_3 \in I_m(req_1)$ (level $k = 1$) and $req_4, req_5 \in I_m(req_3)$ (level $k = 2$).

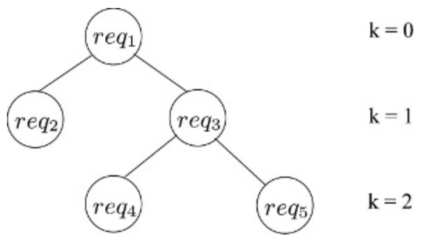


Fig. 3. The *E-tree* for request req_1 from Figure 2

We can construct a consistent state of objects preceding the method call if checkpoints satisfy the *recovery condition* defined as follows:

Definition 2 (The recovery condition.) *Let $e_{a,i}$ be the i-th event in object a. If the state of object a is registered with the following rules the recovery condition is satisfied:*

1. $(e_{a,i+1} = s(req)) \Rightarrow s_{a,i}$
2. $(e_{a,i+1} = r(req)) \Rightarrow s_{a,i}$
3. $(e_{a,i} = s(rep)) \Rightarrow s_{a,i+1}$
4. $(e_{a,i} = r(rep)) \Rightarrow s_{a,i+1}$

Rules 1 and 2 of Definition 2 state that the checkpoint should be registered just before any event of sending and receiving a request. Rules 3 and 4 say that checkpoints should be registered just after sending and receiving replies to requests.

Let's notice that checkpoints in Figure 2 satisfy the recovery condition.

3.3 Consistent State

Determining the consistent state of objects involved in a given remote method call requires that the recovery condition is satisfied and *E-tree* of the given method call is constructed. Then, the consistent state can be determined by examining the *E-tree* beginning from the root level to the lowest level in the following way:

Algorithm 3 (Determining consistent state)

```
1. Add to set $CS$ the checkpoints directly preceding the event of sending request
   in the root node in the E-tree and the event of receiving that request. Add to
   set $O$ the sender and receiver of the request placed in the root node.
2. Initialize variable $k := 0$. It will denote the current level of the E-tree.
3. If $k$ is greater than the depth of the E-tree then stop the algorithm.
4. For each request on level $k$ check if the receiver of the request is in set $O$ and
   if it is not then do:
   (a) add the receiver of the request to set $O$ and
   (b) add the checkpoint directly preceding the event of the receiving this request
       to set $CS$.
5. Increase by 1 variable $k$, i.e., $k := k + 1$. Do step 3.
```

Algorithm 3 examines the *E-tree* and adds checkpoints directly preceding the receive events to set CS. However, the checkpoint is added only if its owner does not belong to set O, i.e., this is the first request of a given receiver. For example, for the *E-tree* in Figure 3 Algorithm 3 will generate the following set $O = \{o_1, o_2, o_3, o_4, o_5\}$. It is correct since all objects from set O are involved in the method call requested in request req_1 (compare the *E-path* of request req_1 in Figure 2). The result set $CS = \{s_{1,1}, s_{2,1}, s_{3,1}, s_{4,3}, s_{5,1}\}$. We can prove that the set of checkpoints, generated by Algorithm 3 assuming that the recovery condition is satisfied, is consistent [7] and as a result it can be used to set states of replica objects in order to test a remote method call.

4 Conclusions

The paper presents the first component of the new methodology called *TestByRep*. *TestByRep* allows for adapting the cyclic debugging technique to distributed object-oriented applications. Determining a consistent state is a key element of *TestByRep*. We showed in this article that it can be done with the concept of hash clocks which enable ordering of events in DOA with respect to dynamic and unknown number of objects in DOA, notions of *E-path* and *E-tree* which provide means for representing a remote method call and allow for determining the consistent state of remote objects participating in the method call.

References

1. J. Huselius, "Debugging Parallel Systems: A State of the Art Report," Tech. Rep. 63, Mälardalen Univ., Dept. of Comp. Science and Engineering, Sept. 2002.
2. M. Ronsse, K. D. Bosschere, and J. C. d. Kergommeaux, "Execution replay and debugging," in *the 4th International Workshop on Automated Debugging*, Aug. 2000.
3. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. of the ACM*, vol. 21, pp. 558–565, July 1978.
4. G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems. Concepts and Design.* Addison-Wesley Longman Limited, 1994.
5. S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," in *Proc. of the USENIX Conference on Object-Oriented Technologies*, June 1995.
6. A. S. Tanenbaum, *Distributed Operating Systems.* Prentice-Hall International, Inc., 1995.
7. M. Sawiska, "Efektywna metoda replikacji dynamicznie powizanych obiektów rozproszonych." Materials for PhD (In Polish), Nov. 2004.
8. M. Sawiska, "Hunting for Bindings in Distributed Object-Oriented Systems," in *Proc. of International Conference on Computational Science – ICCS'2004* (M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3036 of *LNCS*, (Krakow, Poland), Springer, June 2004.
9. M. Raynal and M. Singhal, "Logical Time: A Way to Capture Causality in Distributed Systems," tech. rep., IRISA, Jan. 1995.
10. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, pp. 375–408, Sept. 2002.
11. M. Raynal and M. Singhal, "Logical Time: Capturing Causality in Distributed Systems," *Computer*, vol. 29, no. 2, pp. 49–56, 1996.
12. M. Sawiska, "Testability for Distributed Objects," in *Parallel Processing and Applied Mathematics, 5th International Conference, PPAM 2003, Czestochowa, Poland, Sept. 7-10, 2003. Revised Papers* (R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, eds.), vol. 3019 of *LNCS*, pp. 413–418, Springer, Sept. 2004.
13. M. Neyman, *Metody odtwarzania obliczeń w rozproszonych środowiskach sieciowych.* PhD thesis, Technical Univ. of Gdask, ETI, Feb. 2000. (In Polish).

14. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* The Massachusetts Institute of Technology, 1994.
15. A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom, "Restoring consistent global states of distributed computations," in *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pp. 144–154, ACM Press, 1991.