International Journal of Parallel and Distributed Systems and Networks, Vol. , No. ,

OBSERVATION OF DISTRIBUTED COMPUTATIONS: A REFLECTIVE APPROACH FOR CORBA

L. Duchien* and L. Seinturier†

Abstract

This paper describes some reflective programming techniques to observe a distributed computation in a CORBA environment. First, we propose a new order relation to translate causal dependencies in a distributed program. We generalize Lamport's *Happened before* relation defined for message passing applications, to an object causal relation between distributed events in an environment with synchronous and asynchronous method calls, method synchronizations and variable sharings. Second, we propose a reflective approach to observe this relation. Finally, a tool is provided to display the causal dependencies graph of a distributed run.

Key Words

Causality, CORBA, Reflection, OpenJava, Observation

1 Introduction

With numerous entities distributed over a network, cooperative systems and applications written with CORBA [1] are quite complex and often generate a high volume of communication, numerous concurrent activities, and complex synchronization schemes. Programmers have to master many different software techniques and the design, the development, the debug and the observation of these applications become more and more complex.

In this paper, we focus our attention on the observation of distributed runs in CORBA environments. Like in most existing studies, we do not assume that the system provides a global clock or some perfectly synchronized local clocks. Hence, the observation of a run requires some additional techniques to order distributed events. The partially ordered set approach used by Lamport's *Happened before* relation provides a good solution for such a work. Based on this, numerous studies [2, 3, 4, 5, 6] addressed the issue of the observation of consistent global states. Nevertheless, this relation mainly translates dependencies that are generated by asynchronous communications. First, we can argue that environments such as CORBA rather use synchronous communication schemes. Second, many other sources of dependencies exist in distributed applications. For instance, the synchronization of concurrent methods introduce some dependencies that are not captured by the *Happened before* relation. In this paper, we present an order relation called the object causal order. Its goal is to capture, not only communication dependencies, but also dependencies generated by synchronizations, dynamic creations of threads, and transactionals.

This paper extends previous works [7, 8, 9]. [7] addressed the issue of the observation of a distributed computation for the GUIDE [10] language. [8] was a first study for CORBA/Java environments. [9] introduced reflection, and proposed a *post-mortem* observation tool. This paper tackles the issue of online observation, enhances our system model of distributed events, and reports on some performance measurements. Our target environment is based on the free CORBA ORB JacORB [11], and on the OpenJava [12] reflective language. OpenJava is an extension of the Java language that provides features (i.e. metaclasses) to introspect and to redefine the default semantics of a Java program. We use it to transparently add some code to observe the object causal order.

The paper is divided as follows. Section 2 presents the background and the context of our study. Section 3 defines the object causal order. Next, Section 4 gives the architecture of our tool. Section 5 briefly presents the stamping algorithm and the generated graphs. Section 6 provides some performance measurements, and Section 7 compares our tool with existing works. Finally, Section 8 presents our conclusions and some directions for future works.

2 Background

2.1 Order Relations

The field of order relations for distributed computations has been thoroughly studied. In [13], Lamport introduces a model of sequential processes communicating by asynchronous point-to-point messages. The *Happened before* relation translates causal dependencies in such a model. It is used for instance, for check-pointing, replaying or debugging distributed computations.

Given a set E of local, send and receive events, the *Happened before* order relation, denoted by $\rightarrow$, is the

*CEDRIC-CNAM, 292 rue Saint-Martin, 75141 Paris cedex 03, France; email: Laurence.Duchien@cnam.fr

†Univ. Paris 6, Lab. LIP6, 4 place Jussieu, 75252 Paris cedex 05, France; email:Lionel.Seinturier@lip6.fr

smallest transitive[1] relation satisfying:

- if a and b are events in the same process, and a was executed before b, then $a \to b$,
- if a is a send event by one process and b is the corresponding receive event by another process, then $a \to b$.

The ideas of concurrent events and of consistent cuts can be defined according to this relation (the reader should refer, for instance, to [5] for more details). Most of the existing techniques to compute causal dependencies and consistent cuts use vector stamps [14, 15].

2.2 CORBA

CORBA [1], the standard Object Request Broker from the OMG, proposes an architecture that enables objects to transparently perform requests in a distributed environment. It provides asynchronous (oneway) and synchronous remote method invocations on objects via the ORB. Each object owns an interface described in the Interface Description Language (IDL). On the object server side, the Object Adapter (OA) performs two tasks: (1) it dispatches the incoming method calls to their server objects and, (2) it provides several object activation policies that modify the way methods are executed. For instance, multiple active objects can share the same servant, or only one object at a time can be active on one servant, or each method invocation may be executed by a separate servant.

2.3 Reflection

P. Maes in [16], defines reflection as the ability of a system "to reason and to act upon itself". Reflective programming languages such as CLOS [17], OpenC++ [18], OpenJava [12] or Iguana [19] distinguish two levels of code: the base level that defines the basic functionalities of an application, and the meta level that provides a way to introspect the base level code and to modify its default semantics. The base and the meta levels interact through interfaces and a protocol called a metaobject protocol (MOP for short). The elements of the base level that can be accessed and modified at the meta level are said to be reified. Most existing reflective languages reify method calls. Their default behaviors can then be extended to support for instance, local and remote calls. The extension is transparent to the base level which is unchanged. MOPs can be classified in two categories: compile time and run time. In the former case, the semantics extension defined by the meta levels occurs during the compilation of the program, while in the latter case, it occurs during its execution. Compile time MOPs such as OpenC++ v2 or OpenJava provide better performances, while programs developed with run time MOPs such as CLOS or Iguana are more adaptable and flexible.

In the last few years reflection has become popular in distributed computing as it provides a clear way to handle separation of concerns. Indeed, the numerous functionalities of a distributed program (e.g. communication, concurrency, replication, mobility) can be addressed separately in different meta levels. In this paper, we use reflection to transparently implement an observation service for CORBA applications.

3 Object Causal Order

As pointed out in the introduction, we define the object causal order (denoted by $\to_o$) as an extension of Lamport's *Happened before* relation. The object causal order translates dependencies generated by (1) sequential executions of operations, called local dependencies, (2) synchronous and asynchronous communications, called interaction dependencies, (3) dynamic creations of threads, called thread management dependencies, (4) method synchronizations, called intra-object dependencies, and (5) transactional orderings of read and write operations, called transactional dependencies. Paragraph 3.1 presents our system model. Next, Paragraph 3.2 defines the causal dependencies that we consider in such a system.

3.1 Model of Distributed Events

We consider a system model of multi-threaded objets communicating through a CORBA ORB. We assume that these objects do not share any memory. We also assume that the system does not provide any global clock, nor any perfectly synchronized local clocks. The events that may be generated by such a system are listed below:

1. communication events: objects interact through remote method calls, either synchronous (two ways, blocking), or asynchronous (one way, non blocking). Six events are associated with these operations: method call, send, return, arrival, start, and end. The method call event is the synchronous call of a method. The method return event is the return associated with such a call. The method send event is the asynchronous call of a method. The method arrival event is generated when a method is received on the called object side. Finally, the method start and end events are generated respectively, when a method starts and ends.
2. thread management events: a distributed program is inherently concurrent. It dynamically creates and joins threads. Four events are considered with these operations: thread start, run, end, and join.

[1] i.e. if $a \to b$ and $b \to c$ then $a \to c$

The thread start event is generated when a thread is created. The thread run event is generated when a thread run begins. The thread end event is generated when a thread ends. Finally, the thread join event is generated when a thread join operation is performed.

3. synchronization events: multi-threaded objects may perform synchronizations. For instance, when Java objects are considered, these synchronizations occur when a thread enters a synchronized method, a synchronized block of code, or when *wait* and *notify* methods are called. In our current model, only the first case (synchronized method) is addressed. We leave the other cases for future works. Three events (already mentioned above) are associated with these operations: method arrival, start, and end. Paragraph 3.2.4 defines how dependencies generated by synchronized methods can be detected with these three events.

4. read/write events on shared variables: each of these operations is associated with an event.

3.2 Causal Dependencies

Causal dependencies record order relations between events. These relations are needed when, for instance, a replay service is to be applied to a distributed run. They are also used to construct a logical time for a distributed system. As we do not assume any global clock, this logical clock stamps distributed events. The *Happened before* relation performs such a work, but we argue that other causal dependencies are needed. For instance, consider the case when two executions of a synchronized method are performed concurrently, and when one of these executions is delayed due to the other. If a replay service needs to rerun these executions in the same order, the causal dependency generated by the delay must be recorded.

The object causal order, denoted by $\rightarrow_o$, is the smallest transitive relation satisfying the next five definitions.

3.2.1 Local Dependencies

This first source of dependencies comes from the sequential execution of events within a thread. The definition is the same as in *Happened before.*

Definition 1

- *If e_1 and e_2 are two events that belong to the same thread, and e_1 is executed before e_2, then $e_1 \rightarrow_o e_2$.*

3.2.2 Interaction Dependencies

The interaction source of order translates dependencies created by synchronous and asynchronous communications between local and remote objects. It defines a property similar to Lamport's *Happened-before* relation which assumes that "a message can not be delivered before its sending" [13]. Here, the idea is that each event that is executed before a method call, happens before the execution of the called method. On the same way, each event that is executed after a synchronous method call, happens after the execution of the called method.

Definition 2

- *If e_{sc} is a synchronous method call event, and e_{ma} its corresponding method arrival event, then $e_{sc} \rightarrow_o e_{ma}$.*
- *If e_{ac} is an asynchronous method call event, and e_{ma} its corresponding method arrival event, then $e_{ac} \rightarrow_o e_{ma}$.*
- *If e_{me} is a method end event, and e_{mr} its corresponding method return event, then $e_{me} \rightarrow_o e_{mr}$.*

3.2.3 Thread Management Dependencies

Thread management dependencies create links between a parent thread and its child threads.

Definition 3

- *If e_{ts} is a thread start event and e_{tr} its corresponding thread run event, then $e_{ts} \rightarrow_o e_{tr}$.*
- *If e_{te} is a thread end event and e_{tj} a thread join event waiting for this thread end event, then $e_{te} \rightarrow_o e_{tj}$.*

3.2.4 Synchronization Dependencies

Synchronization dependencies record links between executions of a synchronized method. A synchronized method is granted an exclusive access to its object. Any other thread that tries to access this object will be delayed until the previous execution exits the method. This synchronization scheme introduces a causal dependency between two executions. The dependency can be detected when a method start event can not be performed until an end event associated with the same synchronized method is generated.

Definition 4

- *If e_{me} is a method end event of a synchronized method, and e_{ma} and e_{ms} method arrival and method start events of the same method, and if, in the local object where the methods are performed, e_{me} occurs between e_{ma} and e_{ms}, then $e_{me} \rightarrow_o e_{ms}$.*

3.2.5 Transactional Dependencies

The last source of dependencies comes from the sharing of variables. The basic idea is that read and write operations on a shared variable create dependencies between the threads that perform them. For instance, a read operation can be said to "observe" the effect of the previous write operation. Indeed, the result of the execution would not have been the same if the read had been performed before the write. The transactional relation translates the following dependencies: *read-write*, *write-read*, and *write-write*. As pointed out by the serializability theory (see for instance [20]), a concurrent execution is legal, i.e. is equivalent to a sequential one, if and only if, the transactional dependencies graph deduced from these rules is acyclic.

Definition 5

- *If* e_r *is a read event and* e_w *the next write event on the same variable, then* $e_r \rightarrow_o e_w$.
- *If* e_w *is a write operation and* e_r *the next read operation on the same variable, then* $e_w \rightarrow_o e_r$,
- *If* e_{w1} *is a write operation and* e_{w2} *the next write operation on the same variable, then* $e_{w1} \rightarrow_o e_{w2}$.

4 Observation Service

In this Section we present the prototype of our reflective observation service for CORBA/Java applications. The target ORB is JacORB [11], and the observation is implemented with the OpenJava [12] reflective language.

4.1 Architecture

Our architecture contains two basic components: an observer object, and an observer metaobject (see fig. 1).

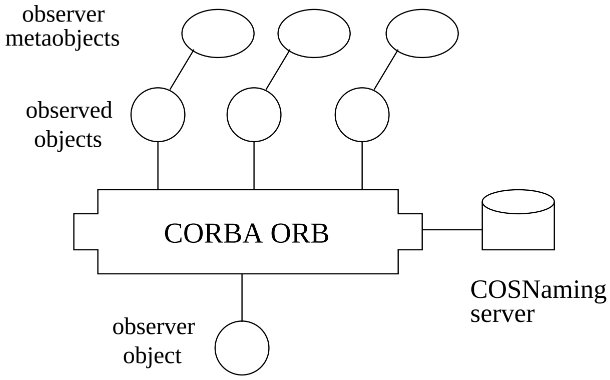

Figure 1: Architecture of the observation service

The observer object is a standard CORBA object. There is one such object for each observed application. It owns an IDL interface with 11 asynchronous methods where each method records one of the events mentioned in Paragraph 3.1. The observer is implemented in Java and stores each received event in a hashtable of vectors. There is one vector per observed object and per observed variable.

An observer metaobject is associated to each application level object that needs to be observed. It reifies elements needed to grab the 11 above mentioned events. Once an event is grabbed, the observer metaobject sends it to the observer object. The binding process between the observer metaobjects and the observer object is kept as simple as possible: the observer registers a well-known name with the CORBA naming service, and each observer metaobject lookups this name. The communication between the observer metaobjects and the observer object is performed by some asynchronous method calls. Unless the CORBA specifications state that the semantics of such calls is "best effort" (i.e. the calls may not be delivered), this mechanism is faster and less intrusive than synchronous method calls.

Transmitted data

When an observer metaobject notifies the observer that an event occured, it transmits the CORBA reference of the observed object, the index of this event in the observee, and the index of the method execution in which this event occurred. Each observer metaobject stores the number of events and the number of method executions that have been generated so far. The observer object needs the first index to reconstruct the object local order, and the second one to associate each event to its method execution (as objects are multi-threaded several executions of the same method may be performed concurrently). Furthermore, for some events, additional parameters are transmitted to the observer object. Table 1 summarizes the event types recorded and their additional parameters.

1. An invocation key is recorded for each method call and method arrival event. This key, which contains the caller object reference, the caller method identifier, and an invocation number, allows the observer object to generate the dependency between the call and the arrival. This key needs to be piggy-backed on each application level method call (indeed, when the method arrival event is generated at the server side, this key needs to be sent to the observer). Our tool provides an IDL preprocessor to automatically perform this task.
2. The parent thread identifier is recorded for each thread run event. This data is needed to generate a dependency between a thread start event and its corresponding thread run event.

Event type	Description	Additional parameters
Method call	A method is called	Invocation key
Method return	A method call is returned	
Method arrival	A method is delivered	Invocation key
Method start	A method begins	
Method end	The method execution ends	
Thread start	A thread start is performed	
Thread join	A thread join is performed	
Thread run	A thread begins	Parent thread id
Thread end	A thread ends	
Read operation	Read of a shared variable	Id and obj ref of the shared variable
Write operation	Write of a shared variable	Id and obj ref of the shared variable

Table 1: Grabbed events

3. Finally, the shared variable identifier and its object reference are transmitted to the observer object each time a read or write operation is performed on a shared variable.

4.2 Observer Metaobjects

4.2.1 OpenJava Meta Features

The code needed to observe the 11 events of Table 1 is automatically added by some OpenJava [12] metaclasses. Like the Java reflection API [21], OpenJava provides a way to introspect the components of a base level program. As shown in fig. 2, the root metaclass of OpenJava is *OJClass*. The `instantiates` keyword is the only modification needed to specify a meta link between a base level class and a metaclass.

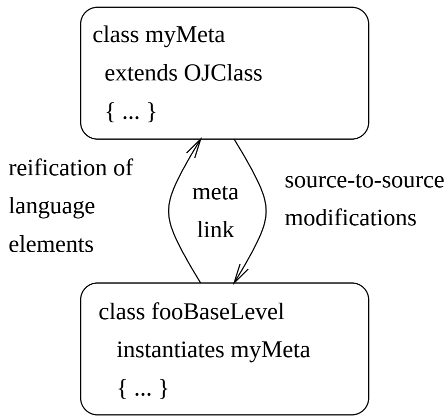

Figure 2: Meta link with OpenJava

Among other things, the interface of *OJClass* (see fig. 3) provides a *getDeclaredMethods* method that returns a description of the base level methods. OpenJava goes a step further than the Java Reflection API and provides a way to add methods or fields (*addMethod* and *addField*), to modify the methods body, or to alter the default semantics of any element in the base level class (*translateDefinition*). Finally, *expand* methods (*expandFieldRead*, *expandFieldWrite* and *expandMethodCall*), are automatically called each time respectively, a field variable is read, a field variable is written, and a method is called. By this way, OpenJava can be seen as a Java language source-to-source translator.

```
class OJClass {
   OJMethod[ ] getDeclaredMethods();
   void addField( OJField field );
   void addMethod( OJMethod field );
   void translateDefinition();
   Expression expandFieldRead( ...  );
   Expression expandFieldWrite( ...  );
   Expression expandMethodCall( ...  );
   ...
}
```

Figure 3: Some selected methods from OJClass

4.2.2 Observation Process

Our main metaclass (*Observer*) is the metaclass of any observed base level class. It extends *OJClass* and customizes its default behavior by, (1) recording the method start and end events (*translateDefinition*), (2) recording the read operation events (*expandFieldRead*), (3) recording the write operation events (*expandFieldWrite*), (4) recording the method call and return events (*expandMethodCall*). The method arrival event is recorded with a wrapper around any synchronized method. The thread related events are recorded with a wrapper class around the standard *java.lang.Thread* class.

The observer metaclass also defines a new keyword: `traced`. It is used as a modifier for base level variables and methods that needs to be traced. By this way, programmers can reduce the volume of trace informations by specifying at compilation time, some relevant elements to trace. Fig. 4 gives the example of an observed class where only fields variables *f1* and *f3*, and method *m2* are traced. Events related to the other variables and methods are not grabbed.

```
class fooToBeObserved
        instantiates Observer {
    traced protected float f1;
    float f2;
    traced static int f3;
    void m1( float x );
    traced int m2( float x );
}
```

Figure 4: Fields and methods tagged with the `traced` modifier are observed

The compile time reflective feature of OpenJava is one of its benefits. As stated in Paragraph 2.3, metaobjects in such languages do not exist during program executions, but only during compilations. The advantage is that there is no execution overhead due to the use of a reflective language. The only overhead introduced comes from the execution of asynchronous method calls to the observer object each time an event is generated.

5 Stamping Process and Graphs

The causal dependencies of a distributed run are computed using vector timestamps. Each element in a vector translates the ordering of events within an activity. In our model, an activity is an application level distributed thread of control that can be stretched on several servers when remote method calls are performed. Activities are created when the application creates threads to carry out new jobs or perform an asynchronous method call. As a distributed application is inherently dynamic, the number of activities, and thus the size of the vector timestamps, are unknown until the end of the run.

Next paragraph describes the way timestamp vectors are constructed. Paragraph 5.2 gives the update rules for these vectors. Finally, Paragraph 5.3 gives an overview of the generated graphs.

5.1 Timestamp Vectors

We define a timestamp vector TE for an event e as $TE_i^j = (te_{i,1}^j, ..., te_{i,n}^j)$, where n is the total number of activities, i the identifier of the activity and j the identifier of the event. This vector is updated each time an event is generated in activity i.

For each shared variable, we manage two vectors TW and TR: $TW_x = (tw_{x,1}, ..., tw_{x,n})$ and $TR_x = (tr_{x,1}, ..., tr_{x,n})$. These are respectively, the timestamp vector of the last write and the timestamp vector of the last read on variable x.

5.2 Rules to Update Timestamp Vectors

This paragraph gives the rules to update timestamp vectors. We assume that the execution order of each activity and the sequential order of read and write operations on each shared variable are known.

Rule 1 *thread start event and asynchronous method call event:* let e, a thread start event or an asynchronous method call event, be the j-th event in activity i. Let q be the identifier of the created thread or method call activity. The q-th element of TE_q^1 is set to 1. To translate the dependency the other elements of TE_q^1 are set to the values of corresponding elements of TE_i^j.

$$\forall q \in [1, ..., p], \forall k \in [1, .., n] \begin{cases} k = q : te_{q,k}^1 = 1 \\ k \neq q : te_{q,k}^1 = te_{i,k}^j \end{cases}$$

Rule 2 *thread join event:* let e, a thread join event, be the j-th event in activity i. Let q be the identifier of the thread, i is waiting for to die. The dependency generated by the last event of thread q must be taken into account.

$$\forall k \in [1, ..., n] \begin{cases} k = i : te_{i,k}^j = te_{i,k}^{j-1} + 1 \\ k \neq i : te_{i,k}^j = Max(te_{q,k}^{last}, te_{i,k}^{j-1}) \end{cases}$$

Rule 3 *method start event:* let a, a method start event, be the j-th event in activity i. If the considered method is synchronized, and if there exists a method end event e_{me} whose timestamp vector is TE_q^p, and a method arrival event e_{ma} of the same method, and if for the local objects where the events are generated, e_{me} occurs between e_{ma} and e, then the i-th element of TE_i^j is increased, and its other elements are set to the maximum of the corresponding elements of TE_i^{j-1} and TE_q^p.

$$\forall k \in [1, ..., n] \begin{cases} k = i : te_{i,k}^j = te_{i,k}^{j-1} + 1 \\ k \neq i : te_{i,k}^j = Max(te_{i,k}^{j-1}, te_{q,k}^p) \end{cases}$$

Rule 4 *read event:* let e, a read event on variable x, be the j-th event in activity i. The i-th element of TE_i^j is increased and its other elements are set to the maximum of corresponding elements of vectors TE_i^{j-1} and TW_x. TR_x is also updated to record the read dependency for the next write operation.

$$\forall k \in [1, ..., n] \begin{cases} k = i : te_{i,k}^j = tr_{x,k} = te_{i,k}^{j-1} + 1 \\ k \neq i : te_{i,k}^j = tr_{x,k} = Max(te_{i,k}^{j-1}, tw_{x,k}) \end{cases}$$

Rule 5 *write event:* let e, a write event on variable x, be the j-th event in activity i. The i-th element of TE_i^j is increased and its other elements are set to the maximum of corresponding elements of vectors TE_i^{j-1}, TW_x and TR_x. TW_x records the timestamp of the last write operation and TR_x is cleared to avoid redundant transitive dependencies.

$$\forall k \in [1,...,n] \begin{cases} k = i : te_{i,k}^j = tw_{x,k} = te_{i,k}^{j-1} + 1 \\ k \neq i : te_{i,k}^j = tw_{x,k} = \\ \qquad\qquad Max(te_{i,k}^{j-1}, tw_{x,k}, tr_{x,k}) \\ tr_{x,k} = 0 \end{cases}$$

Rule 6 *other events:* let E be the j-th event in activity i. We increase the i-th element of TE_i^j.

$$\forall k \in [1,...,n] \begin{cases} k = i : te_{i,k}^j = te_{i,k}^{j-1} + 1 \\ k \neq i : te_{i,k}^j = te_{i,k}^{j-1} \end{cases}$$

5.3 Graphs

Based on the information sent by the observer metaobjects, a causal dependencies graph is generated online by the observer object. It is then displayed with the VGJ [22] tool which is a graph viewer application. VGJ provides a framework to plug customized graph manipulation algorithms. We designed such an algorithm for our observation process: it instanciates the CORBA observer object, records the events, and generates the graph. The graph is updated as new events are sent to the observer object, and as some new dependencies are detected. When a new activity is detected, either through a thread start event or an asynchronous method call, the timestamp vector size of all previously received events is increased by one.

Our tool[2] provides a panel with buttons to control the display of the graph. It can be paused, resumed and moved forward or backward. Note that this panel only controls the display, not the computation itself. Even if the display is paused, the computation keeps running. Fig. 5 gives a screen snapshot of our tool. A text description of the graphs can be generated in the GML [23] markup language (this is a built-in feature of VGJ).

6 Performance Measurements

This Section presents some performance analysis conducted with our tool. We provide some evaluations of the overhead introduced by the observation process. The testbed platform includes two Sun Sparc Ultra 5 on a 10 Mbits/s Ethernet network. Both machines are running Solaris 5.7, version 1.1.6 of Sun JDK, JacORB 1.0b7, and OpenJava 1.0. Table 2 summarizes our results.

The first set of tests evaluates the cost of grabbing events. As stated in Paragraph 4.1, each distributed event is sent by an observer metaobject to the observer object through an oneway method call. In the worst case, the cost of one sending is 1.7 ms, and 1.5 ms in the best case. This difference comes from the fact that the amount of data sent differs from one type of event to another one. These results correspond to the mean time of 10 sequences of 1000 grabbed events. The same test performed when the observer and the observee are colocated on the same machine, gives respectively 1.5 ms and 1.3 ms. This shows that most of the time is spent in the client stub and server skeleton of the CORBA environment, not in the network.

The second test evaluates the cost of piggy-backing parameters to the application level method calls. As stated in Paragraph 4.1, we piggy-back an invocation key to record dependencies between method calls and method arrival events. Our tests show that a standard CORBA twoways method call between remote machines takes 2.6 ms. The overhead introduced by the extra parameter is 0.3 ms. These results correspond to the mean time of 10 sequences of 1000 twoways method calls.

7 Comparison with Other Works

This Section reviews some existing observation tools, and compares them to our work.

[24] propose some tools to observe parallel applications developped with the PVM library. Their approach uses queries as a device for searching the state space, visual presentation techniques adapted from program animation, and provides the ability to navigate through the state space using some visual interactions. Both this work and ours use Lamport's *Happened before* relation. Nevertheless, we think that the target audience of our tool is broader as we take advantage of CORBA and Java which are widely used for distributed applications.

[25] describes an IIOP protocol analyzer based on *tcpdump*. It displays IIOP messages that are sent and received by CORBA objects. Inprise's AppCenter [26] performs quite a similar task at the request level (a CORBA request may be split in several IIOP messages). Both tools are limited to protocol related events, and does not consider, as our tool does, JVM related events (such as thread creations), or application related events (such as method synchronizations).

The MAScOTTE [27] and GoodeWatch [28] projects deal with the observation of CORBA ORBs. MAScOTTE defines a management information base to mon-

[2] Our tool can be downloaded from our Web page: *http://www-src.lip6.fr/homepages/Lionel.Seinturier/RCO/*.

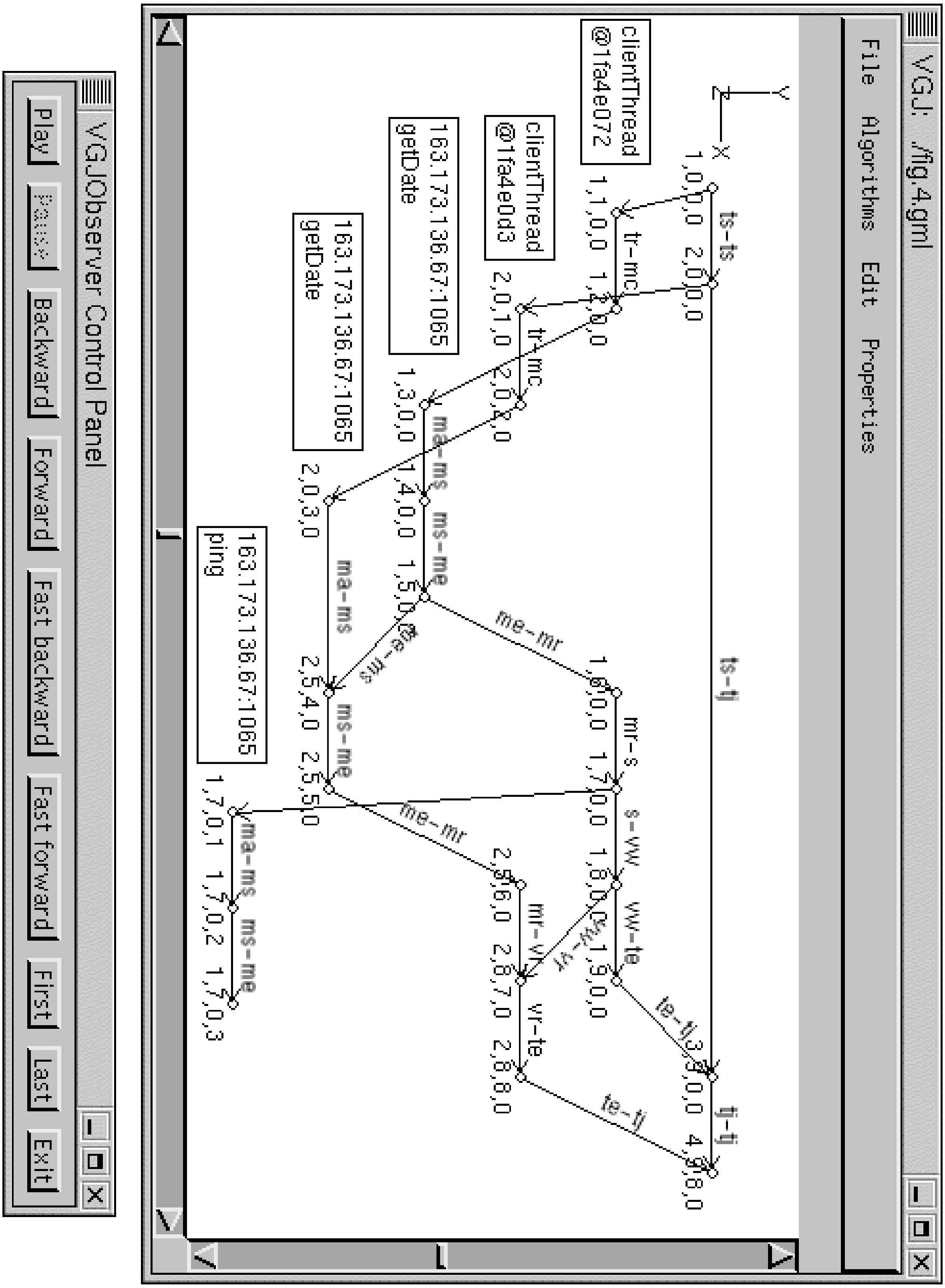

Figure 5: Screen snapshot of the observer tool

		Overhead	Percent of time spent	
			network	CORBA ORB
Event grabbing	Worst case	1.7 ms	12 %	88 %
	Best case	1.5 ms	13 %	87 %
Piggy-backing		0.3 ms		

Table 2: Performance measurements

itor the activity of core CORBA components. GoodeWatch provides mechanisms to grab events occuring at the ORB level. Our tool goes a step further and, not only grabs ORB related events, but also provides a smart display through the detections of causal dependencies between these events. As far as we know, none of the above mentioned tools perform such a work.

8 Conclusion

This paper presents a causality relation called the object causal order, for distributed applications in a CORBA environment. This relation extends Lamport's *Happened before* relation by (1) considering both synchronous and asynchronous communications (Lamport only considers asynchronous ones), and (2) incorporating dependencies generated by communications, method synchronizations and variable sharings (Lamport only considers communications). By this way, we think that the object causal relation provides a better understanding of the semantics of distributed applications.

The second main point of our paper is that the relation is observed by taking advantage of the features of a reflective language. As stated in Section 4, our target CORBA ORB is JacORB [11] and our target reflective language is OpenJava [12]. We developed some OpenJava metaclasses to transparently add the code needed to record our causal dependencies. These metaclasses reify events related to method calls and processings, thread management and read/write operations on shared variables. Events generated by the application are sent by these metaclasses to a global observer. Next, we define vector timestamps for the generated events and we provide an algorithm to compute the causal dependencies graph. Finally, our tool, which is an extension of the existing VGJ [22] viewer, displays this graph. It is updated online as new events are sent to the observer.

Several extensions can be considered for this work. First, algorithms could be added to check global predicates (with techniques described for instance in [29] and [30]). Second, some tools could be developed to filter and to analyze more precisely the traces. By the same way, notions such as abstract event [31], which are non empty sets of primitive events, could also be investigated in order to reduce the volume of trace data.

References

[1] OMG. The common object request broker: Architecture and specification. OMG, February 1998.

[2] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transac. on Computers*, 36(4):471–482, April 1987.

[3] B. Miller and D. Choi. Breakpoints and halting in distributed programs. In *Proc. of the 8th Intl. Conf. on Distributed Computing Systems*, pages 316–323, 1988.

[4] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transac. on Computer Systems*, 3(1):63–75, February 1985.

[5] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report SFB 124 - 15/92, Univ. of Kaiserslautern, December 1992.

[6] M. Ahuja, T. Carlson, A. Gahlot, and D. Shands. Timestamping events for inferring affects relation and potential causality. In *Proc. of COMPSAC'91*, pages 606–611, 1991.

[7] P. Placide, L. Duchien, G. Florin, and L. Seinturier. A consistent global state algorithm to debug distributed object-oriented applications. In *Proc. of AADEBUG'95*, May 1995.

[8] L. Duchien and E. Jeury. Observation in CORBA Java applications. In *Proc. of the Session on Coordination at PDPTA'99*, June 1999.

[9] L. Duchien and L. Seinturier. Reflective observation of CORBA applications. In *Proc. of IASTED PDCS'99*, pages 311–316, November 1999.

[10] R. Balter and al. Architecture and implementation of GUIDE, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[11] G. Brose. JacORB: Implementation and design of a Java ORB. In *Proc. of DAIS'97*, September 1997. *http://www.inf.fu-berlin.de/˜brose/jacorb.*

[12] M. Tatsubori and S. Chiba. *OpenJava 1.0 API and Specification*. Programming Language Lab., Univ. of Tsukuba, 1998. *http://www.softlab.is.tsukuba.ac.jp/˜mich/ openjava*.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[14] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computing Conf.*, February 1988.

[15] F. Mattern. Virtual time and global states in distributed systems. In *Proc. of the Intl. Conf. on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[16] P. Maes. Concepts and experiments in computational reflection. In *Proc. of OOPSLA'87*, pages 147–155, December 1987.

[17] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] S. Chiba. A metaobject protocol for C++. In *Proc. of OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 285–299, October 1995.

[19] B. Gowing and V. Cahill. Meta-object protocols for C++: The Iguana approach. In *Proc. of Reflection'96*, 1996.

[20] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[21] Sun Microsystems. *Java Core Reflection, API and Specification*, February 1997. *http://www.javasoft.com*.

[22] C. McCreary. *Drawing Graphs with VGJ*. Auburn Univ., 1998. *http://www.eng.auburn.edu/department/cse/ research/graph_drawing/graph_drawing.html*.

[23] M. Himsolt. GML: A portable graph file format. Univ. Passau, 1997. *http://www.fmi.uni-passau.de/Graphlet/GML/ gml-tr.html*.

[24] D. Hart, E. Kraemer, and D. Roman. Interactive visual exploration of distributed computations. In *Proc of the 11th Intl. Parallel Processing Symposium*, April 1997.

[25] C. Treanor. IIOP Protocol Analyser. *http://www-rst.int-evry.fr/˜defude/analyseur-iiop.html*, 1999.

[26] Inprise. Inprise AppCenter. *http://www.inprise.com/appcenter*, 1999.

[27] Introduction to MAScOTTE, Esprit Project 20804. White paper, May 1997. *http://www.esrin.esa.it/MAScOTTE*.

[28] C. Gransart, P. Merle, and J.M. Geib. GoodeWatch: Supervision of CORBA applications. In *ECOOP'99 Workshop on Object-Orientation and Operating Systems*, June 1999.

[29] C. Chase and V. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11, 1998.

[30] V.K. Garg. Observation and control for debugging distributed computations. In *Proc. of AADEBUG'97*, 1997.

[31] T. Basten, T. Kunz, J. Black, M. Coffin, and D.J. Taylor. Vector time and causality among abstract events in distributed computations. *Distributed Computing*, 11:21–39, 1997.

Biographies

Laurence Duchien is an associate professor in the Computer Science Department at the Conservatoire National des Arts et Métiers, Paris, France, since 1990. She received her Ph.D degree in computer science from University Pierre et Marie Curie, Paris, France, in 1988. Her research interests include distributed algorithms for cooperative applications, design methodologies, and proof systems for distributed object-oriented applications.

Lionel Seinturier is an associate professor in computer science at the University Pierre et Marie Curie, Paris, France, since 1999. In 1998, he worked as a research engineer at France Télécom R&D center. He received his Ph.D degree in computer science from Conservatoire National des Arts et Mtiers, Paris, France, in 1997. His research interests include distributed algorithms, reflective systems, and middleware platforms.