Hunting for Bindings in Distributed Object-Oriented Systems[⋆]

Magdalena Sławińska

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

Abstract. The paper examines the problem of finding a group of objects that are involved in a certain transitive relation. It is especially important when a group of related objects has to be identified, for example for monitoring. The article defines static and dynamic binding relations between objects in a distributed object-oriented system. It also presents an architecture for catching these relations since current operating systems do not support such mechanisms. In the paper the algorithm for finding bindings of a given object is described.

1 Introduction

In distributed object-oriented systems objects are scattered over the network. One of the main design goals of distributed systems is to provide different types of transparency for users, e.g., location or replication transparency [1][2]. However, sometimes such transparency can be very uncomfortable for testers and programmers, for example during debugging.

This paper presents the framework for finding objects that are involved in a certain transitive relation. For instance, consider the situation when a tester wants to examine method m of object o_1. However, method m invokes method *m1* of object o_2. It means that object o_1 is in a *binding relation* with object o_2. The problem complicates when the so-called *native* and *foreign* objects are considered [3]. The former are objects with the full access to the source code by testers while the latter are those ones for which testers have only on-line access to objects' methods but no access to the source code.

Finding bindings among distributed objects is necessary when programmers want to isolate a group of *bound objects*, for example in order to limit the number of monitored objects only to relevant ones. It is also important in program replay or recovery [4]. The problem is similar to causality tracking, however it has a different flavor [5].

This paper is organized as follows: in Section 2 the model of distributed object-oriented system is presented. Section 3 introduces two important relations between objects: static and dynamic binding relations. In Section 4 the

⋆ Funded in part by the State Committee for Scientific Research (KBN) under grant 4-T11C-004-22

© Springer-Verlag Berlin Heidelberg 2004

architecture of the system for identifying bound objects is shown. Section 5 presents the algorithm for identifying bound objects and finally in Section 6 the paper is concluded.

2 The System Model

A distributed object-oriented system is a finite set of objects performing a given task. It is denoted by $S = \{o_1, o_2, ..., o_n\}$, where $n = 1, 2, 3, ...$. There is a finite set of hosts denoted by $H = \{h_1, h_2, ..., h_k\}$, where $k = 1, 2, 3, ...$. Objects are located in hosts. One object can be located in only one host but many objects may reside in one host. Objects and hosts are identifiable.

In contrast with classical procedural distributed systems like PVM (*Parallel Virtual Machine*) [6] or MPI (*Message Passing Interface*) [7] where processes communicate with each other by message passing, objects cooperate with other objects through method invocations. In fact, remote method invocation mechanisms only wrap the message-passing stuff. Messages are still sent and received over the network, however, it is done by special software entities automatically generated by compilers like *stubs* and *skeletons* [1][8]. They are responsible for object binding, (un)marshaling parameters and results, issuing method on a target object. In fact, considering the lower level of abstraction, a remote method invocation which returns a certain result consists of the following steps: (1) a client object sends request *req* for the method invocation of a server object, (2) the server object receives request *req*, (3) the server object carries out request *req*, (4) the server object sends reply *rep* to request *req*, (5) the client object receives reply *rep*. So invoking a remote method with a return value consists of four communication actions: two *sends* and two *receives*. Two kinds of messages are exchanged during a method invocation: *requests* (denoted by *req*) and *replies* (denoted by *rep*). Requests as well as replies are *messages* (denoted by *mesg*). A message consists of an identifier, a source object, a target object and contents. In case of requests the contents contains an identifier of a method to be invoked and parameters (if any). The contents of replies comprises a method identifier that has been invoked and a return value (if any). Figure 1 schematically shows a method call in terms of sending and receiving requests and replies. Please notice that it is also possible to model local invocations on a given object in the send/receive request/reply semantics (although it is inefficient). There are three objects in Figure 1: o_1, o_2 and o_3. Horizontal lines represent time and arrows denote messages. Symbols $s(\cdot)$ and $r(\cdot)$ stand for send and receive events, respectively. Subsequent events are numbered in the context of a given object (e_i). An identifier of a message consists of a source object identifier (unique in the system) and a number of an event in the context of the source object (unique in the object). Message identifiers contain a type of a message (reply or request).

For simplicity reasons it is assumed that all methods of an object are public and can be invoked remotely.

A sequence of events of a given object forms its history (h). For instance in Figure 1: $h_{o_1} = (e_1, e_2)$, $h_{o_2} = (e_1, e_2)$ and $h_{o_3} = (e_1, e_2, e_3, e_4)$. A history of an

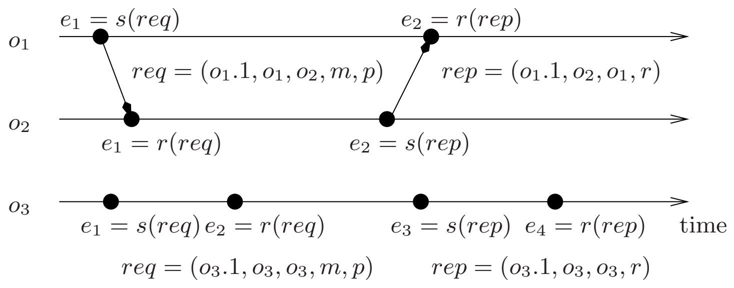


Fig. 1. Method invocations in the send/receive request/reply semantics in distributed object-oriented systems.

object is available for other objects. It may be implemented by interception of communication events. For example *interceptors* are defined in CORBA [8]. The Eternal system, which provides fault tolerance for distributed objects, intercepts method invocations in order to assure replica consistency [9].

3 Relations among Distributed Objects

Relations among objects concern the static or dynamic point of view.

3.1 Statically Bound Objects

In case of native objects, it is possible to deduce certain connections among objects by examining their source code.

Definition 1 (The static binding relation) *Object o_1 is in the "static binding" relation with object o_2 if the following conditions are satisfied:*

SB1. Object o_1 is a client of object o_2.
SB2. Condition SB1 can be deduced from the source code of object o_1.
SB3. Object o_1 does not create object o_2.

Notation $o_1 \looparrowright_s o_2$ means that object o_1 is in the "static binding" relation with object o_2.

The simplest static binding relation is presented in Example 1.

Example 1

```
class A {
  public void m1() {}
  public void m2() {
      oB.m1();
  }
  public void m3() {
      C c = new C();
      c.m1();
  }
};
```

```
class B {
  public void m1() {}
  public void m2() {}
};
```

```
class C {
  public void m1() {}
  public void m2() {}
  public void m3() {}
};
```

Let's assume that the system consists of two objects: *oA* and *oB* which are instances of classes `A` and `B`, respectively. It is clear from the pseudo-code of class `A` (i.e. condition SB2 is satisfied) that *oA* is a client of *oB* (*oA.m2* calls *oB.m1*). It means that SB1 is satisfied. Since there is no instruction of *oB* creation in the scope of *oA*, also SB3 is satisfied. It implies that $oA \looparrowright_s oB$. Let's notice that although SB1 and SB2 are satisfied in case of objects *oA* and *c*, $oA \not\looparrowright_s c$ since SB3 is violated.

Relation $\looparrowright_s$ is not symmetrical. In spite of $oA \looparrowright_s oB$, $oB \not\looparrowright_s oA$ since condition SB1 is not satisfied. Relation $\looparrowright_s$ is transitive.

Example 2

```
class A {                  class B {                  class C {
  public void m1() {}        public void m1() {}        public void m1() {}
  public void m2() {         public void m2() {         public void m2() {}
      objB.m1();               objC.m1();               public void m3() {}
  }                          }                        };
};                         };
```

Let's assume that there are three objects *objA*, *objB* and *objC* which are instances of classes `A, B` and `C` defined in Example 2. It is easy to notice that $objA \looparrowright_s objB$ and $objB \looparrowright_s objC$ since SB1, SB2 and SB3 are satisfied. It means that if *objA* is to exist, also *objB* should exist and if *objB* is to exist also *objC* should exist what implies that since *objA* should exist, *objC* should exist. The presented deduction is performed on the source code analysis (SB2 is satisfied). In fact, *objA* is an indirect client of *objC* (SB1 is satisfied). Also *objC* is not created by *objA* (SB3 is satisfied too).

The static binding relation is suitable for finding hypothetical relationships among objects. For instance, in Example 1 it is possible that method *oA.m2* will be never invoked in real execution, i.e., *oA* will never be a client of *oB*. Relation $\looparrowright_s$ only indicates that a given binding is probable but it does not guarantee that it really happens.

3.2 Dynamically Bound Objects

In case of foreign objects, the source code is unavailable. In order to identify bindings among foreign objects, or to identify bindings that happened in the past, it is necessary to analyze their histories.

Definition 2 (The dynamic binding relation) *Object o_1 is in the dynamic binding relation with object o_2 if the following conditions are satisfied:*

DB1. In the history of object o_1 exists event e which is a sending of a request of a method invocation of object o_2.
DB2. If o_1 is in the dynamic binding relation with object o_2 and o_2 is in the dynamic binding relation with object o_3 then o_1 is in the dynamic binding relation with object o_3.

Notation $o_1 \looparrowright_d o_2$ means that object o_1 is in the dynamic binding relation with object o_2.

For example in Figure 1 $o_1 \looparrowright_d o_2$ since $e_1 \in h_{o_1}$ and $e_1 = s(o_1.1, o_1, o_2, m, p)$. Let's notice that $o_3 \looparrowright_d o_3$ since $e_1 \in h_{o_3}$ and $e_1 = s(o_3.1, o_3, o_3, m, p)$. In comparison to relation $\looparrowright_s$, relation $\looparrowright_d$ is transitive by definition (condition DB2). Please notice, that DB2 is not a property since relation $\looparrowright_d$ may concern the same objects but different methods and transitivity will be not satisfied.

Relation $\looparrowright_d$ extends $\looparrowright_s$ since sometimes the source code of objects is unavailable (foreign objects) or even if it is available it is impossible to find out with which objects the interaction will be performed (e.g. a dynamic vector of object references).

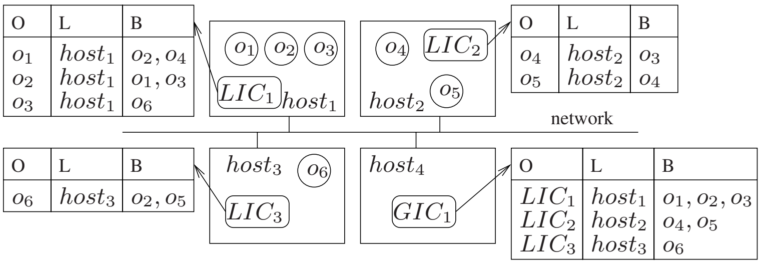
3.3 Bound Objects

It is useful for further consideration to define more general binding relation.

Definition 3 (The binding relation) *Object o_1 is in the binding relation with o_2 if object o_1 is in the static binding relation with o_2 or object o_1 is in the dynamic binding relation with o_2, i.e.,* $(o_1 \looparrowright o_2) \iff (o_1 \looparrowright_s o_2) \vee (o_1 \looparrowright_d o_2)$.

Since relation $\looparrowright_s$ is transitive and $\looparrowright_d$ is transitive by definition then relation $\looparrowright$ defined as a logical sum is transitive.

Binding relations can be represented by directed graphs where vertices depict objects in the system and directed edges denote binding relations (e.g. Figure 4).



O	L	B
o_1	$host_1$	o_2, o_4
o_2	$host_1$	o_1, o_3
o_3	$host_1$	o_6

O	L	B
o_4	$host_2$	o_3
o_5	$host_2$	o_4

O	L	B
o_6	$host_3$	o_2, o_5

O	L	B
LIC_1	$host_1$	o_1, o_2, o_3
LIC_2	$host_2$	o_4, o_5
LIC_3	$host_3$	o_6

Fig. 2. The architecture for finding bound objects.

4 The Architecture for Finding Bound Objects

Figure 2 presents the architecture for finding objects which are involved in the binding relations defined in the previous section. There are three important entities in Figure 2: *Global Information Center (GIC)*, *Local Information Center (LIC)* and *Object-Location-Bindings (OLB)* tables. Both GIC and LIC objects keep the so-called OLB tables where they store necessary information. Column L has the same meaning for GIC-OLB or LIC-OLB tables. It shows a location of an object from column O. In case of a GIC-OLB table, column O contains LIC objects registered in a given GIC object whereas column O of a LIC-OLB table shows objects registered in a given LIC object. For example, in Figure 2 GIC_1 keeps information about LIC_1, LIC_2 and LIC_3, while LIC_3 stores information about object o_6. Column B has different meaning for GIC-OLB and LIC-OLB tables. A GIC-OLB table contains data about objects registered in a given LIC object. However, in case of LIC objects column B indicates the set of objects which a given object from column O is in relation $\looparrowright$ with. For instance, in Figure 2 from GIC_1-OLB it is clear that two objects are registered in LIC_2 (o_4 and o_5). However, table LIC_3-OLB indicates that $o_6 \looparrowright o_2$ and $o_6 \looparrowright o_5$.

GIC as well as LIC objects are responsible for keeping current information in their OLB tables. The presented architecture assumes that it exists a special layer (not depicted in the figure) that is responsible for recording histories of objects to the log. In order to keep a GIC-OLB table up-to-date different strategies can be used, e.g., *push* (changes are pushed to GIC objects), *pull* (GIC objects pull information from LIC ones) or *mixed* (pull or push when necessary) models [10].

The hierarchical structure of the architecture assures scalability [1]. In order to improve performance and availability GIC objects may be replicated [1][2][9].

In order to find out a list of objects bound with a given object a graph of bindings in the system can be constructed.

5 Constructing a Graph of Bindings

A graph of bindings in a system can be constructed with Algorithm 1. The result of Algorithm 1 is array `A[n][n]` of object bindings, where `n` is the number of objects in the system. Function `index(ObjectId)` assigns `ObjectId` in the system to a column number in `A` (see Figure 3). Function `object(idx)` (reverse to `index(·)`) assigns `idx` in `A` to an object id. In array `A` two values are possible: 0 and 1. If value A[i][j] equals 1 it means that $object(i) \looparrowright object(j)$, otherwise $object(i) \not\looparrowright object(j)$.

Algorithm 1 (Constructing a matrix of bindings)
```
1. Update the GIC-OLB table (if necessary).
2. Construct array A[n][n] and fill it with 0 values.
3. For each LIC in the GIC-OLB table do:
   (a) Get column O of the LIC-OLB table
   (b) For each object o in column O do:
           Get relevant values from column B of the LIC-OLB table
              For each value b from column B do:
                 A[index(o)][index(b)] = 1
```

The result of Algorithm 1 is the matrix of bindings in the system. For the system in Figure 2 the result of the algorithm is shown in Figure 3. In order to get a directed graph it is necessary to transform array `A` by, for example, Algorithm 2. The obtained graph is presented in Figure 4. In order to generate a list of bound objects Algorithm 3 may be applied.

object(·) $\uparrow$ index(·) $\downarrow$

$$\begin{array}{cc|c|c|c|c|c|c|c}
 & & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & \\
 & & 0 & 1 & 2 & 3 & 4 & 5 & \\
\hline
\text{bindings} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & o_1 \\
 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & o_2 \\
 & 2 & 0 & 1 & 0 & 1 & 0 & 0 & o_3 \\
 & 3 & 1 & 0 & 0 & 0 & 1 & 0 & o_4 \\
 & 4 & 0 & 0 & 0 & 0 & 0 & 1 & o_5 \\
 & 5 & 0 & 0 & 1 & 0 & 0 & 0 & o_6 \\
\end{array}$$

table A

Fig. 3. The matrix of bindings in the system in Figure 2.

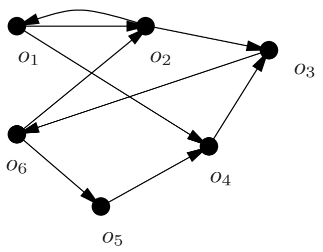


Fig. 4. The transformed matrix from Figure 3.

Algorithm 2 (Transforming A2G)

```
1. Put n vertices.
2. Label each vertex according to function object()
3. For each column in A do:
     For each value in column do:
       if(A[i][j] == 1) then {draw an arrow from object(i) to object(j)}
```

Algorithm 3 (Finding a list of bound objects)

```
1. A;                     // up-to-date table of bindings (Algorithm 1)
   Vector V;              // the dynamic list of bound objects
   Set    S;              // a result set of bound objects
   int I = index(O);      // for what object -- O we have to look for
   boolean isRemoved;     // indicates if an element was removed from V
2. do{
     isRemoved = false;   // nothing was removed from vec
     makeBoundObjectList(V, S, A, I); // see point 5
     if( V.size() > 0){
       I = V[0];          // take index of object to be checked
       V.remove(0);       // remove it from from vector V
       isRemoved = true;} // indicate that the length of V was decreased
   }while ((V.size() != 0) || isRemoved );
3. check in A if O is in the relation with itself; update S if necessary
4. S contains a list of indexes which object O is bound with
5. Procedure makeBoundObjectList(V, S, A, I):
     for (i = 0; i < A[I].length; i++)
       if( A[I][i] == 1 ) then  // object(I) in relation with object(i)?
         if ( S.add(i) ) then   // check if elem. i in S and add if not
           V.add(i);            // we must check bindings of object(i)
                                // so remember it in V
```

The result obtained from Algorithm 3 on table A from Figure 3 for finding bindings of o_1 is $S = \{1, 3, 2, 5, 4\}$ what implies (after object(·)) $\{o_2, o_4, o_3, o_6, o_5\}$.

The main idea of Algorithm 3 is to investigate `A` in order to find objects that are transitively bound with a specified object. Procedure `makeBoundObjectList` looks for objects that are in relation $\looparrowright$ with a given object (`I`) and if it finds some, set `S` is updated (please notice that this is a *set* so no duplicates are allowed). Next, since the added object may be in the relation with other objects (and since it has just been added to the set, so it was not checked earlier) it must be added to `V` for further investigation. From vector `V` elements are systematically removed as they are checked for relations in `makeBoundObjectList()`.

6 Conclusions

The paper describes the algorithm for finding objects involved in the transitive binding relation with a given object. It is especially important if a tester wants to identify a group of objects. The article presents the framework architecture for maintaining information about bindings among distributed objects. The algorithm makes use of OLB tables, GIC and LIC services. It constructs the 2-dimensional table of bound objects. Having such a table it is possible to find out all objects that are bound with a given object. The situation complicates in the case of foreign objects since it is practically impossible to deduce relationships. It implies that special architectures for logging relevant information is necessary. The presented algorithms have been implemented in a prototype tool in order to verify the concepts in practice.

References

1. G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems. Concepts and Design.* Addison-Wesley Longman Limited, 1994.
2. A. S. Tanenbaum, *Distributed Operating Systems.* Prentice-Hall International, Inc., 1995.
3. M. Sławińska, “Testability of Distributed Objects,” in *Proc. of the 5-th International Conference on Parallel Processing and Applied Mathematics*, Springer-Verlag, 2003. (to appear).
4. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “A survey of rollback-recovery protocols in message-passing systems,” *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.
5. L. Alvisi, K. Bhatia, and K. Marzullo, “Causality tracking in causal message-logging protocols,” *Distributed Computing*, vol. 15, no. 1, pp. 1–15, 2002.
6. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM:Parallel Virtual Machine: A Users’ Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
7. Message Passing Interface Forum, ed., *MPI: A Message-Passing Interface Standard.* Message Passing Interface Forum, June 1995.
8. OMG, *Common Object Request Broker Architecture: Architecture and Specification, v3.0.* `http://www.omg.org`, December 2002.
9. P. Narasimhan, L. Moser, and P. M. Melliar-Smith, “State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects,” in *Proc. of the IEEE Int. Conf. on Depend. Syst. and Net.*, IEEE Computer Society Press, 2001.
10. OMG, *Event Service Specification, v1.1.* `http://www.omg.org`, March 2001.