Iranian Journal of Science & Technology, Transaction B, Engineering, Vol. 30, No. B6, pp 763-773
Printed in The Islamic Republic of Iran, 2006
© Shiraz University

CMI: A METHOD TO CONFIGURE CORBA REMOTE CALLS IN SUPPORT OF FT-CORBA FAULT-DETECTION MECHANISMS*

M. SHARIFI** AND H. SALIMI

Dept. of Computer Engineering, Iran University of Science and Technology, Tehran, I. R. of Iran
Email: msharifi@iust.ac.ir

Abstract– Collocated CORBA objects that reside in the same address space can benefit from special local calls which can be performed without ORB intervention. This type of invocation can be particularly beneficial to fault detection mechanisms defined in FT-CORBA specifications. According to FT-CORBA, a group of objects, namely fault detectors, periodically monitor the status of replicated objects in the system to make sure that they are alive. In cases where a fault detector object is collocated with some of the objects that are monitored by this detector, direct invocations can improve the performance of fault detection mechanisms. All the known available methods for direct calls to collocated CORBA objects are flawed either with unnecessary pre-checks that are performed before each invocation, or with violation of the remote call semantics (like bypassing ORB and POA). In addition, as this paper shows, in some cases the default pre-checks that are performed before a call are not sufficient, and none of the available methods allow the applications to perform domain-dependent pre-checks or to only override the default ones. *CMI (Configurable Method Invocation)* is a new method that allows the pre-checks to be selected before each direct call in order to avoid investigating unnecessary conditions. Furthermore, it allows the developers to make application-dependent pre-checks or override the existing ones before each call. To achieve these two properties, we have changed the CORBA IDL compiler in such a way to generate a special code in addition to the code for collocation-safe stubs. This extra code permits the developer to manipulate the execution of pre-checks which must be performed before each call. Implementation results of our method show a 41% reduction of communication overheads in a fault detection mechanism compared to the standard approach. The possibility of checking a user-defined pre-check before each direct call is also shown.

Keywords– Collocation, FT-CORBA, fault-detection, standard and direct calls

1. INTRODUCTION

Although the earliest reliable CORBA (Common Object Request Broker Architecture) implementations have been introduced during the last decade, and despite the adoption of the fault tolerant CORBA (FT-CORBA) [1] standard by Object Management Group (OMG), CORBA is still not considered the preferred platform for building dependable distributed applications [2].

The FT-CORBA specification divides the fault management process into three activities: (1) fault detection, (2) fault notification and (3) fault analysis, and proposes appropriate mechanisms for each of these activities. For example, for fault detection activities the so-called fault detector objects are used, in which a sentinel object monitors the state of replicated objects.

There are mainly two methods for the detection of faults in FT-CORBA which include: (1) PUSH-based monitoring and (2) PULL-based monitoring. In the PUSH-based approach, every object informs its liveness to the fault detector object periodically. On the other hand, in the PULL-based approach, the fault detector object pings the other replica objects in order to make sure that the other objects are alive.

*Received by the editors October 16, 2005; final revised form November 5, 2006.

**Corresponding author

An important point about the fault detection mechanisms introduced by FT-CORBA specifications is that nothing is mentioned about the communication type between the fault detector and the replica objects. For example, in PUSH-based methods, all invocations can be defined as one-way calls. Also, in PULL-based methods, direct call techniques can be applied in order to achieve better performance.

Several methods have already been proposed in which two collocated objects can communicate, but because of the limitations that exist in these methods, they cannot be used as a communication technique for connecting a fault detector to replica objects. The reason for this problem is that none of the proposed methods are capable of performing user-defined pre-checks in a way that collocated objects can communicate in a CORBA-compliant way as well. In this paper, we extend the existing methods so that direct calls are allowed to be configured, and at the same time, the calling is kept CORBA-compliant.

The remainder of this paper is organized as follows. Section 2 provides a minimal overview of the architecture of CORBA remote invocations required in our discussions. Section 3 describes some notable related works. Section 4 presents some of the important limitations of the existing methods. Section 5 presents our proposed approach. Section 6 illustrates some of the experimental results of our approach, and finally, Section 7 concludes the paper.

2. OVERVIEW OF CORBA REMOTE CALLS

The architecture of CORBA remote calls is designed in such a flexible way that it can be easily configured for different purposes. There are four main elements involved in a remote call [3]: (1) ORB, (2) POA, (3) POA Managers and (4) Servants.

a) ORB

ORB (Object Request Broker) is responsible for connecting the client and server objects. ORB resides on top of the operating system as a transparent layer and delivers the requests to the right objects, regardless of their locations, platforms and also the languages in which those objects have been implemented.

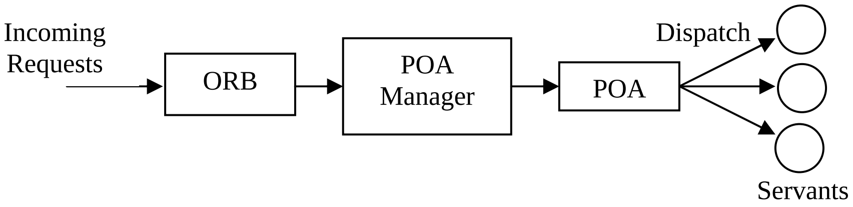

Fig. 1. The request flow of a remote invocation in CORBA

b) POA

The main purpose of a POA (Portable Object Adaptor) is to connect the abstract concept of a CORBA object and the concrete representation of that object's behavior in the form of a servant. In other words, a POA can be seen as a mapping tool that redirects incoming requests to the relevant objects in the server's memory.

Each POA has seven policies that are associated with that POA when it is created (and remain in effect without change for the lifetime of each POA). The policies control aspects of the implementation techniques that are used by servants using that POA, such as the threading model and the persistent state of object references.

c) POA manager

A POA manager acts as a request entrance gate that redirects the flow of received messages to one or more associated POAs. Conceptually, a POA manager represents a transport endpoint (such as a host–port pair for TCP/IP). A POA is associated with its POA manager when the POA is created; thereafter, the POA manager for a POA cannot be changed. A POA manager can be in one of the following four possible states:

1) **Active.** This is the normal state in which the POA manager passes all of the incoming requests to the target POA.
2) **Holding.** In this state, the POA manager holds requests in a queue. Once the POA manager enters the active state, it passes all of the requests to their destination POAs.
3) **Discarding.** Incoming requests are rejected with a TRANSIENT exception. This exception indicates to the client that the request cannot be delivered right now, but that it may work if retransmitted again later.
4) **Inactive.** Requests are rejected; however, instead of raising an exception, the POA manager indicates to the client that the connection to the server is no longer usable. Depending on how the client is configured, this may result in an attempt by the client to locate a new instance of the server.

d) Servants

A Servant is a language-specific object that receives the requests. It receives the requests from its associated POA, which is assigned to it at its creation time.
The general request flow into a server is shown in Fig. 1. Note that the diagram only represents a conceptual view. In the implementation, requests are not physically passed in this way for efficiency reasons.

3. RELATED WORKS

To put our proposed approach into context, first The ACE ORB (TAO) is described. After that, the previous attempts at reducing communication overheads between CORBA objects which reside in the same address space are discussed.

a) TAO

TAO (The ACE ORB) [4] is an ORB endsystem that includes the network interface, communication protocol, operating system, and CORBA middleware components and features shown in Fig. 2. TAO is based on the CORBA reference model, with the following improvements which are designed in order to overcome the drawbacks of legacy ORBs in support of real-time applications:

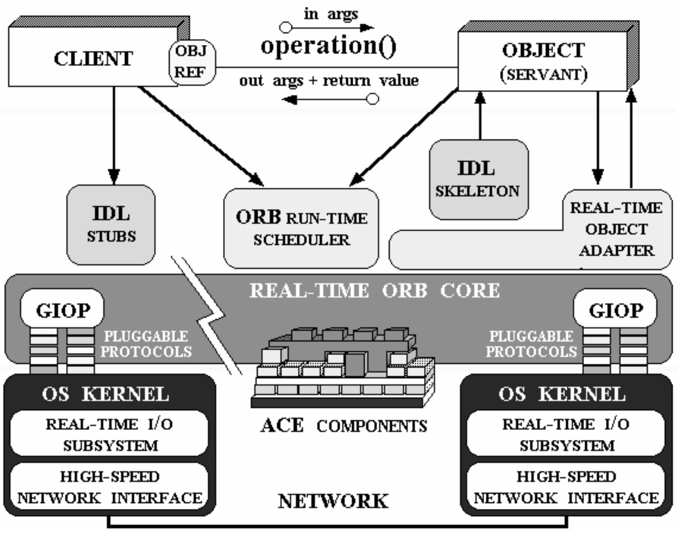

Fig. 2. Components in the TAO Real-time ORB [5]

Real-Time IDL Stubs and Skeletons: In addition to marshaling and demarshaling of operation parameters, TAO's Real-time IDL (RIDL) stubs and skeletons are designed to ensure that application timing deadlines are fulfilled.

Real-Time Object Adapter and ORB Core: In addition to associating servants with the ORB and demultiplexing incoming requests to servants, TAO's Object Adapter (OA) implementation dispatches servant operations in accordance with various real-time scheduling strategies.

ORB QoS Interface: TAO's QoS interface is designed to map real-time processing requirements to ORB endsystem network resources. Common real-time processing requirements include end-to-end latency bounds and periodic scheduling deadlines. Common ORB endsystem/network resources include CPU, memory, network connections and storage devices.

Real-time I/O Subsystem: TAO's real-time I/O subsystem performs admission control and assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be guaranteed.

b) Collocation Optimization

In this section, previous attempts at reducing communication overheads between CORBA objects which reside in the same address space are discussed under four headings: (1) TAO strategies, (2) CCM language mapping, (3) development of local CORBA components and (4) local ORB-like services.

TAO Strategies: Collocation optimizations for CORBA can be considered as a technique to transparently optimize communication overheads when clients and servants reside in the same address space. Collocation optimizations on TAO [5-7] have mainly focused on two different techniques: *direct* and *standard*.

In *direct* collocation technique, all requests are forwarded directly to the servant object. Thus, the other intermediary elements are not involved at all. Performance results have shown that *direct* collocation invocations of CORBA objects are almost comparable to virtual method invocations of ordinary C++ objects. But because of omitting some of the important elements like POA and POA manager, this technique is no longer CORBA compliant.

The *standard* collocation technique uses a so-called *collocation-safe* stub to handle operation invocations on a collocated object. Collocation-safe stubs perform a set of pre-checks before dispatching the request to the target object. These pre-checks are performed in support of keeping the invocation CORBA compliant, but these activities cause a significant amount of overhead [6].

Although the *direct* approach implemented in TAO yields better performance, there are some problems because of not using the ORB and POA functionality. These problems, as discussed in Section 4, imply that this technique is not suitable to be deployed in FT-CORBA fault detection mechanisms.

CCM Language Mapping: The CORBA Component Model (CCM) Specification [8], contains language mappings which define local interfaces. These interfaces are used to implement local components. The local keyword was added in the CCM specification and can appear in an IDL before the interface keyword. This keyword indicates that the interface can only be implemented by objects in the current process.

A local interface is implemented by extending the *CORBA::LocalObject* class. IDL attributes and operations are implemented in the same way as a regular servant object. But the difference is that the invocations that are targeted to an object which implements a local interface will not pass through the ORB and other intermediary elements.

This approach extremely decreases the communication overhead between a component and its facets residing in the same address space. However, as will be discussed in Section 4, this approach can not guarantee a trusted communication mechanism for connecting a fault detector and a replica object.

Development of Local CORBA Components: The concept of *Local Component Adapter Concept* (LCAC) is introduced in [9]. This approach separates the application code from the implementation of the CORBA component logic. For every IDL definition of a CORBA interface or component, a corresponding interface in the native implementation language is defined. Adapter classes provide CORBA mappings, and link the implementation of business logic to the CCM component. By taking advantage of the adapter concept, the developers can implement local components without a CORBA shell. Using a local path for connecting components significantly reduces the communication overhead, but as some of the previous research show, this approach will not remain CORBA compliant because of failure to use some of the CORBA utilities (e.g. POA and POA manager).

Local ORB-Like Services: In our previous work, *Local ORB-Like Services* are presented [10]. This approach tries to connect collocated components with local ORB like services support. In this approach, a unit inside each container is responsible for handling communication between components within or outside the container. Local requests are passed to the local components without ORB involvement. Local or remoteness of a request is determined from the IOR of the called component which has been logged by the relevant special unit upon the creation of the component in its container.

It has been shown that the above approach greatly reduces the communication overhead between a component and its collocated components residing in the same address space. However, in the case of fault detection systems, it is strongly required to have the components connected using some intermediate elements like POA and ORB. So, as discussed in greater detail in Section 4, this approach cannot be appropriate as a communication mechanism to be used between the fault detector and replica objects.

4. LIMITATION OF AVAILABLE METHODS TO BE USED IN FT-CORBA FAULT DETECTION MECHANISMS

All available methods for decreasing the unnecessary overheads of collocated object calls can be categorized into two main groups. The first group includes those techniques in which the remote calls do not remain CORBA-compliant. In these techniques, some intermediate elements like POA are bypassed, so the performance of invocations increases dramatically. The *direct* technique which is used by TAO and the others described in Section 3b are some examples of this approach. On the other hand, the second group includes those techniques that try to respect the principles of the CORBA remote call. A well-known technique in this category is the *standard* call technique mentioned in Section 3b. Based on our knowledge, this technique is the only method that can connect two collocated objects directly, and at the same time tries to keep the invocations CORBA-compliant.

The techniques in the first group cannot be used for connecting a fault detector to a replica object. The reason is that the replica object may encounter a failure due to the loss of its connection to ORB. Since the ORB is not involved in this type of communication, the failure will not be detected at all.

The techniques in the second group are more akin to our proposed approach, but are flawed with two drawbacks: (1) performing unnecessary pre-checks and (2) disability of overriding the default pre-checks. These drawbacks are discussed in greater detail in Sections 4a and 4b.

a) Performing Unnecessary Pre-Checks

One of the weak points that makes the second group of methods inappropriate is the number of unnecessary investigations that are performed before each invocation. Going through all these pre-checks incurs a non-trivial amount of overhead [5]. The invocation of an operation via a collocation-safe stub must undergo the following checks:

1) The server ORB (which may or may not be the ORB used by the invoking client) has not been shutdown.
2) The POA managing the servant still exists.
3) The POA manager of this POA is queried to make sure invocations are allowed on the POA's servants.
4) The POA's policies, *e.g.*, the Thread Policy are respected.
5) Interceptors are invoked at the proper interception points.
6) The invocations are not re-directed elsewhere.
7) One way method calling semantics must be preserved.

What is important to note about these steps is that in many cases investigating all of the mentioned steps is unnecessary. For example, in a direct call from a fault detector object to a replica object in a PULL-based monitoring scheme, it is not necessary to check Steps 4 and 7, and sometimes Steps 5 and 6. Also in a PUSH-Based monitoring scheme, there is no need to check Steps 3 and 4, and Steps 1 and 2 must be performed in another way. Table 1 tabulates the necessary pre-checks before any invocation under these two monitoring schemes.

Table 1. Pre-checks that must be performed under the two monitoring schemes before any call from a fault detector to a replica object or vice versa

Required Investigations, Needed to be Performed before Each Invocation	PULL-Based Monitoring	PUSH-Based Monitoring
1. Checking Servant's ORB Liveness	Yes, because any failure on servant's ORB must be reported as a fault.	No, instead, the status of the client ORB must be checked.
2. Checking Servant's POA Liveness	Yes, because any failure on servant's POA must be reported as a fault.	No, instead, the status of the client POA must be checked.
3. Checking Servant's POA Manager Liveness	Yes, because any failure on servant's POA manager must be reported as a fault.	No, because if POA manager is on a non-active state, the pre-checks must be performed
4. Checking Servant's POA Thread Policy	No, because existing two threads in a servant, one of whom is the thread of fault detector object, is not dangerous (Call to is_alive() method does not change the state of the servant object).	Depends, this case depends on the fault detector object, i.e. is the replica object in this case.
5. Calling all Registered Interceptors	Depends, it depends on whether any interceptors are registered or not.	Depends, for the same reason mentioned for PULL-Based monitoring.
6. Considering Re-Directed Invocations	Depends, it depends whether re-directed invocations are enabled on ORB or no.	Depends, for the same reason mentioned for PULL-Based monitoring.
7. Respecting Oneway Invocation Semantics	No, invocations from fault detector object to replica object can not be oneway.	Yes, because all invocations from replica object to fault detector can be oneway.

b) Disability of Overriding the Default Pre-checks

As is shown in Table 1, in some cases the default pre-checks to be performed before an invocation must be changed to fulfill some of the domain-dependent requirements. For example, in a PUSH-based monitoring scheme, instead of checking the status of the servant's ORB, the status of the client’s ORB must be checked. The reason for this change is that the replica object must send a fault report to the fault detector when its ORB is not alive.

5. OUR PROPOSED APPROACH

As discussed in Section 4, the techniques in which the current research is focused on have the following drawbacks:

1) Some of them do not keep the invocations CORBA-compliant.
2) Those that respect the CORBA remote call principles (like the TAO’s standard approach), implicate a number of unnecessary pre-checks.
3) None provides an application with embedded domain-dependent investigations.

To solve the above mentioned problems, a CMI approach is proposed. This approach is based on TAO’s *standard* call. As depicted in Fig. 3, in addition to the TAO’s *Collocation-Safe Stub (CSS)*, an extra part, namely *Extended Collocation-Safe Stub (ECSS)*, is designed which inherits from the main stub. In this case, the client ORB creates an instance of ECSS when it wants to wrap an object reference, instead of using an instance of CSS.

Many ORBs need some information to set a few policies up before running. As an example, our version of ORBacus [11] needs a configuration file to be placed in the same path of the client application. From this file, the ORB acquires the necessary information to be initialized, like its concurrency model. The path of this file is passed to the orb_init() method as a parameter. In this file, we also provide the ORB with the information about using the appropriate stub. In the case of using ECSS, we determine which pre-checks should be investigated in this stub.

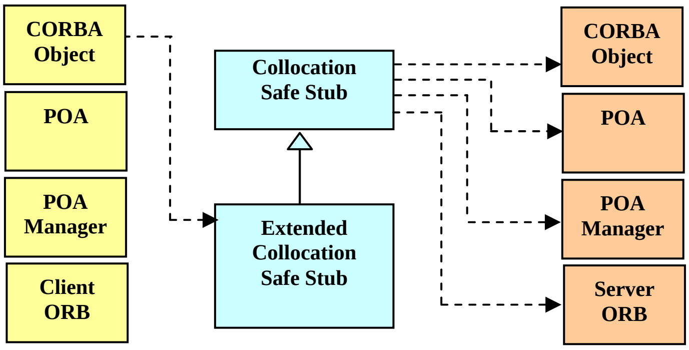

Fig. 3. Communication architecture of two collocated objects using Extended Collocated-Safe Stub (ECSS)

The code for ECSS is generated by a CORBA IDL compiler. We have changed the IDL compiler in such a way to generate a code for this extended part. ECSS is used to override the default pre-checks of CSS. In this stub, the developer can either put his own pre-checks or use the default ones prepared at the main stub.

In order to provide the ability for selecting the pre-checks that must be performed before each call, another strategy is employed. As shown in Fig. 4, the CSS is equipped with a utility called the *Configuration Manager (CM)*. The CM is responsible for providing the CSS with the checks that must be performed before each invocation. The CM interface is implemented in such a way to return the calling parameters to CSS. The default implementation of this interface reads the settings from a file. Other implementations may realize this interface, and read the configurations from somewhere else, e.g. a system registry.

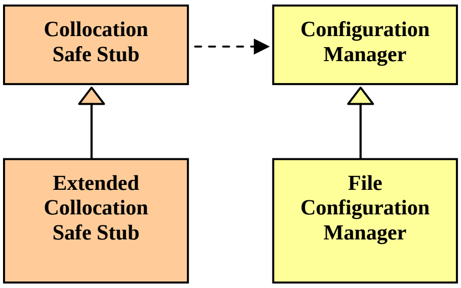

Fig. 4. CMI's configuration management strategy

To clarify our proposed approach, some parts of the generated code for the *is_alive* method at the *IPULLMonitorable* interface is depicted below.

```
CORBA::Boolean
OBDirecStubImplEx_IPULLMonitorable::is_alive(
CORBA::AbstractBase_ptr p)
{
  PortableServer::POA_ptr   _col_poa_;
  if (_col_poa_ = OBGetCollocated_poa(p))
  {
    if (!_OBCheck_Prerequisites())
            OB::RaiseFailureException();

    return ((POA_IPULLMonitorable*)(_ob_servant_))->is_alive();
  }
  else
    return OBDirectStubImpl_IPULLMonitorable::is_alive();
}

CORBA::Boolean
OBDirectStubImplEx_IPULLMonitorable::_OBCheck_Prerequisites()
{
  unsigned long _direct_call_config_;
  _direct_call_config_ = _ob_config_mgr_base -> get_config();

  if (_direct_call_config_|CHECK_ORB)      if (!_ob_check_orb())
    return false;
  if (_direct_call_config_|CHECK_POA)      if (!_ob_check_poa())
    return false;

  // Other Checking for POA Manager, Interceptors
  // and … will be placed here.

  Return true;
}
```

According to the given code, there are two main classes in the generated code: *OBDirectStubImpl_IPULLMonitorable* and *OBDirectStubImplEx_IPULLMonitorable.* The former is collocation-safe stub, and the later class plays the role of extended collocation-safe stub.

As illustrated, the collocation check in the *OBDirectStubImplEx_IPULLMonitorable::is_alive* method is performed in order to see if the target object is collocated or not. If the target object is collocated, first the prerequisites are checked. If the prerequisite checks are successful, the target object is called directly; otherwise a system exception is raised.

For checking the prerequisites of a call, a method, namely *_OBCheck_Prerequisites*, is provided. This method gets the pre-checks from the *CM* and makes sure that the CORBA remote call semantics are preserved.

6. PERFORMANCE EVALUATION

To measure the performance gain from CMI's optimization, we ran client and server objects, O1 (as a fault detector) and O2 (as a replica object) in the same process. As depicted in Fig. 5, O2 realizes an interface, namely *IMainObject*. This interface inherits from the *IPULLMonitorable* interface defined in the FT-CORBA specification. The only method of the *IPULLMonitorable* interface is *is_alive*, which is used by fault detectors to monitor the state of replica objects.

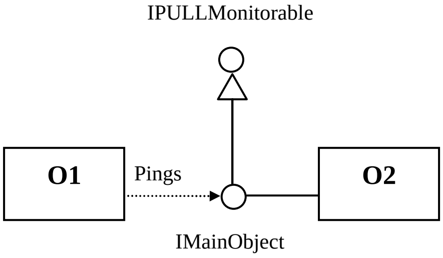

Fig. 5. The class diagram of our benchmark program

The platform used to benchmark the test program was an Athlon 1.8 MHz running Microsoft Windows XP with SP1. The test program was developed using ORBacus (Version 4.4.1) and the codes were compiled by Microsoft Visual C++ 6.0 compiler.

To compare performance systematically, the test program was run with the CMI strategy, the *standard* collocation strategy, and the *direct* collocation strategy, as well as with no collocation optimization, i.e., using remote stubs. To compare the performance gain of collocation optimizations to the optimal performance, we also measured the time to perform the same tasks by making direct virtual function calls on the target servant. In all of these tests, the *is_alive* method was the target function which was called by O1.

We ran our benchmark program 100 times and in each run, for every invocation technique, we made 10000 calls from the fault detector to the replica object. Using all the measured data, we calculated the average calls per second for each invocation type. Figure 6 shows the performance improvement, measured in *calls-per-second*. With the CMI optimization, we obtained a performance improvement of 41% compared to the case when the calls were made using the standard strategy. It should be noted that although other approaches such as virtual and direct show better performance than CMI, they are not suited to fault detection mechanisms because of violation of remote call semantics. A feature-wise comparison of our approach with other approaches is briefed in Table 2.

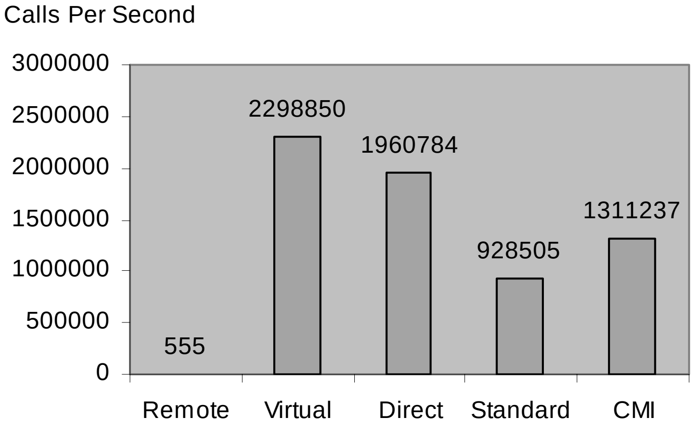

Fig. 6. The results of CMI's performance optimization

Table 2. A feature-wised comparison of our proposed approach

	Collocation Optimization	Environment	Preserving CORBA Remote Call Semantics	Configuring the Pre-Checks	Overriding of Pre-Checks
Collocation Optimization	Yes	TAO	Yes	No	No
CCM Language Mapping	Yes	CCM Spec.	No	No	No
Local CORBA Components	Yes	MicoCCM	No	No	No
Local ORB-Like Services	Yes	MicoCCM	No	No	No
CMI	Yes	ORBAcus	Yes	Yes	Yes

7. CONCLUSION AND FURTHER WORKS

In this paper we presented the limitations of the current methods for supporting CORBA collocated object calls. As a case study we showed why these methods are not suitable for connecting a fault detector to a replica object. A new approach was presented based on TAO's standard calling. This approach allows the collocated objects to be connected without entailing customary CORBA overheads, and at the same time permits the invocation settings to be fully configured or customized. Both features are handled by an extended stub called *Extended Collocation-Safe Stub*, which is generated by a special IDL compiler. Experimental results have shown a 21% performance improvement in cases where a fault detector and its entire replica objects are collocated.

To continue the current research, we are planning to work on the following topics in future:

1) Creating a set of policies modeled after CORBA, ORB, and POA policies that could be applied on the client to configure the checks that are done for the collocated case.
2) Introduce efficient techniques in order to omit the overhead of the collocation check before each call to an object.
3) Extend the PULL-based fault detection mechanism in such a way that could tune its call period proportional to the loss of oneway calls in a real environment.

4) Designing mechanisms in which objects that are involved in a fault detection mechanism could change their selected scheme dynamically.

REFERENCES

1. Object Management Group, (2001). Fault Tolerant CORBA (Final Adopted Specification). OMG Technical Committee Document, formal/01-12-29.
2. Felber, P. & Narasimhan, P. (2004). Experiences, strategies and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers, Vol. 53*(5).
3. Object Oriented Concepts, Inc. (2000). *CORBA/C++ programming with ORBacus*. Student Workbook, Version 1.0.5, http://www.ooc.com
4. Center for Distributed Object Computing, TAO, (2002). A high-performance real-time object request broker. Washington University www.cs.wustl.edu/~schmidt/TAO.html
5. Pyarali, I., O'Ryan, C., Schmidt, D.C., Wang, N., Kachroo, V. & Gokhale, A. (1999). Applying optimization patterns to design of real-time ORBs, in *Proceedings of the* 5th *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX.
6. Schmidt, D.C., Wang, N. & Vinoski, S. (1999). Object interconnections: Collocation optimization for CORBA, SIGS C++ Report.
7. Schmidt, D. C., Wang, N. & Levine, D. (2000). *Optimizing the CORBA Component Model for High-Performance and Real-Time Applications*, Middleware, New York.
8. Object Management Group, (2000). CORBA Component Model. Technical Report 01-11-03.
9. Teiniker, E., Mitterdorfer, S., Kreiner, C. & Kovacs, Z. (2002). Local components and reuse of legacy code in the CORBA component model. EUROMICRO 2002, Dortmund, Germany.
10. Sharifi, M., Rahmani, A.T., Rafe, V., Momeni, H. (2004). CORBA components collocation optimization with local ORB-like services support. Lecture Notes in Computer Science (LNCS), No. 3291, Springer-Verlag, 1143-1154.
11. IONA Technologies Co., ORBacus 4.1.1, http://www.orbacus.com