Architecture of ROAFTS/Solaris: A Solaris-Based Middleware for Real-Time Object-oriented Adaptive Fault Tolerance Support

Eltefaat Shokri and Patrick Crane - SoHaR Inc., USA
K. H. (Kane) Kim and Chittur Subbaraman - University of California, Irvine, USA

Abstract

Middleware implementation of various critical services required by large-scale and complex real-time applications on top of COTS operating system is currently an approach of growing interests. Its main goal is to enable significant reduction in the complexity of application system design and implementation by separating the concerns of the application designer for the application functionality from the concerns for application-independent system issues. This paper presents the middleware architecture named the Real-time Object-oriented Adaptive Fault Tolerance Support (ROAFTS) and a prototype implementation ROAFTS/Solaris realized on top of both a COTS operating systems, Solaris, and a COTS CORBA-complaint ORB, Orbix. ROAFTS supports distributed real-time applications, each structured as a network of Time-triggered Message-triggered Objects (TMO's), and the TMO is a major extension of a conventional object for use in hard real-time applications. The major components of ROAFTS include a TMO support manager for supporting the execution of TMO's, a generic fault tolerance server, and a network surveillance manager (NSM) which provides the generic fault tolerance server with fast fault detection notices. The generic fault tolerance server and the NSM themselves have been structured as TMO's. A discussion on an effective use of CORBA standards for moderate-precision real-time applications to run on COTS operating systems is also presented.

Keywords: CORBA, real-time objects, fault tolerance, middleware, COTS, adaptive, TMO

1. Introduction

Many emerging large-scale safety-critical applications are highly distributed, and must operate in highly dynamic environments with varying timeliness, functional, and reliability requirements [Kim92, Hur96]. The robust design and implementation of these complex systems is a difficult challenge and designers of such systems wait eagerly for significant advances in distributed systems design methodologies, system engineering tool environments, and operating system services. The following principles may assist system designers in realization of such complex systems:

- **Middleware Implementation**: In order to reduce the complexity in the design and the implementation of emerging larger-scale applications, it is imperative to maximally separate the concerns for *application functionality* from the concerns for other system attributes such as "guaranteeing required reliability" and "low-level inter-component communication facilities". In other words, subsystems providing common services such as distributed fault tolerance support can be built as a separate layer (residing on top of the operating system kernel) represented by a minimal set of well-defined interfaces to the applications [Sho97].
- **Real-Time Object-oriented Structuring**: Experiences in designing large applications over the past decade have demonstrated that object-oriented design and implementation techniques have good potentials for reducing the design complexity and increasing the robustness and maintainability of the resulting systems. Recently, there have been efforts [Kim94a, Kim97b] to extend the object-oriented structuring to effectively deal with the main concerns of the real-time application developers (i.e., timing characteristics for objects). We strongly believe that the use of real-time object structuring approaches is essential for the design of reliable, reusable, extensible, and maintainable applications.
- **Use of Standards and COTS Components**: The reliability of Commercial-of-the-self (COTS) hardware/software components, built in conformance with widely-accepted standards, is dramatically increasing, thus making these components attractive building-blocks for critical applications. One of the well-established software component standards of growing importance is Object Management Group's CORBA, a location-transparent language-independent

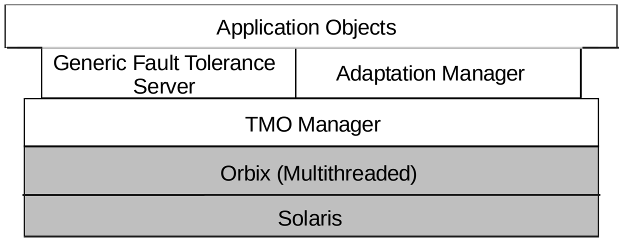

Figure 1. ROAFTS components

inter-object communication architecture [OMG95]. CORBA's Object Request Broker (ORB) provides a high-level abstraction for communications among objects residing in different hosts.

- **Adaptability**: Since some emerging applications must operate in highly dynamic environments and the available execution resources may change dynamically, it is highly useful to make the applications as well as the underlying execution systems to adapt to dynamic changes in the application environments, the available resources, or the application requirements.

This paper presents the architecture called the *Real-time Object-oriented Adaptive Fault Tolerance Support* (**ROAFTS**) middleware, and a prototype implementation realized on top of both a COTS operating system, Solaris, and a COTS CORBA-compliant ORB, Orbix.

Figure 1 depicts major components of ROAFTS. Solaris was chosen as the base operating system because it provides some real-time support with its real-time threads mechanism. Iona's Orbix is a multi-threaded lightweight CORBA 2.0-compliant ORB. The Time-Triggered Message-Triggered Object (TMO) structuring scheme, formulated in recent years [Kim94a, Kim97b], is in our view one of the most natural extensions of the conventional object-oriented design and implementation techniques and it possesses the following important characteristics: (a) in addition to traditional *service methods* (methods called by client objects), each TMO may also have *time-triggered methods* (activated when the real-time clock reaches specific points in time); (b) new natural concurrency constraints and the rules for confining the actions to occur at fixed times to be within time-triggered methods are incorporated to greatly reduce the complexity in making design-time guarantees of the timeliness of object services. Several experiments on the TMO structuring, conducted in a variety of application contexts ranging from military applications to factory automations [Kim94a, Kim97b], validated the proposition that the TMO-based real-time system design offers a rigorous and yet efficient/economic way to develop complex real-time systems in easily understandable forms. The TMO Manager layer in the ROAFTS middleware supports the applications structured as a network of TMO's. Correct and timely execution of the TMO's according to their specifications is the main responsibility of the TMO Manager.

There are some fault management elements which are application-specific (such as a non-trivial and realistic acceptance test designed to check the reasonableness of application states or outputs). Therefore, it is useful to isolate application-independent elements from application-specific ones. In the ROAFTS middleware, the Generic Fault-Tolerance Server provides fault-tolerance services as a set of C++ abstract classes from which the customized objects can be constructed.

The Adaptation Manager recognizes any major changes in either the environment or the available execution resources and reconfigures the system to cope with the current state of the system/environment.

This paper begins in Section 2 with a discussion on the major features of CORBA and the essential features of an effective COTS operating system. The major design and execution schemes supported by the ROAFTS middleware are discussed in Section 3. Section 4 provides the detailed architecture of the middleware. Section 5 discusses our experiences in implementing a CORBA-compliant prototype of ROAFTS on top of Solaris and our conclusions.

2. Platform characteristics

2.1 CORBA

The underlying philosophy of CORBA is to provide a common architectural framework for distributed computing across heterogeneous hardware platforms, operating systems, and inter-node communication protocols. The core of the CORBA architecture is the Object Request Broker (ORB), a mechanism that supports remote object activation and inter-object communication.

The function of the ORB is to deliver requests from clients to server objects and return output values (if any) back to the clients. Clients do not need to know where in the network server objects reside, how they communicate, or how they are implemented. This implementation transparency is achieved by separating object interfaces from object implementations.

Under CORBA, a remote request from an object to a remote server object must be passed to the ORB residing in the client host. The request will then be sent to the ORB in the server host. The server ORB then locates the requested object and initiates the execution of the requested method. When the execution of the requested method is completed, the result will be transferred to the client object through the server ORB and the client ORB. It is important to note that the client object issuing the request does not have any control on the timing of the series of the actions to be taken for the execution of the remote methods. This will be discussed in Section 4.

2.2 Essential features of a candidate COTS operating system

The COTS operating system on which an efficient ROAFTS middleware is to be built, should provide mechanisms for the following capabilities:

- *Lightweight concurrency unit*: To impose minimal overhead for switching among different "units of concurrent execution" within the application, lightweight application-level concurrency units should be supported by the operating system. This, however, may be provided in different forms. For example, Solaris offers library-implemented application threads that are partially transparent to the kernel and are managed by the thread-management library. Other operating systems such as Windows NT and VxWorks support different types of threads or tasks.
- *High-precision timer interrupt*: The operating system should also allow the application to define an interval timer which can stimulate a computing action (such as generating a signal or reactivating a specific process) when the preset waiting time expires. Since all the time-triggered actions in our implementation model will be enacted by the timer, the timing of the signal generation must be accurate and the gap between the timer's generation of a signal and the activation of a relevant action must be within a very small bound. The granularity of the clock supporting the timer, which differs in different operating systems, is a major factor to consider in selecting a platform for a specific application domain.
- *Real-time threads*: Here, the term real-time thread refers to a generic construct which has the following characteristics: (i) the delay it experiences in accessing a resource should not exceed a predetermined tight bound, (ii) it is forced to share the execution engine with any other process or thread (even another real-time thread), (iii) it releases the execution engine only voluntarily or via a "terminate" signal from the kernel, (iv) when a system service call is issued inside a real-time thread, the system call inherits the real-time characteristics of its issuer thread. These characteristics guarantee timely initiations of the actions to be taken by a real-time thread.
- *Efficient control of support processes*: The COTS operating systems designed for typical business data processing environments create various long-life background support processes (often called daemons) for performing day-to-day management and housekeeping activities. Since these processes, which are transparent to the application designer, steal the resources from the applications to carry out their activities, their actions at unpredictable times may make timely executions of application activities difficult and thus must be controlled appropriately.

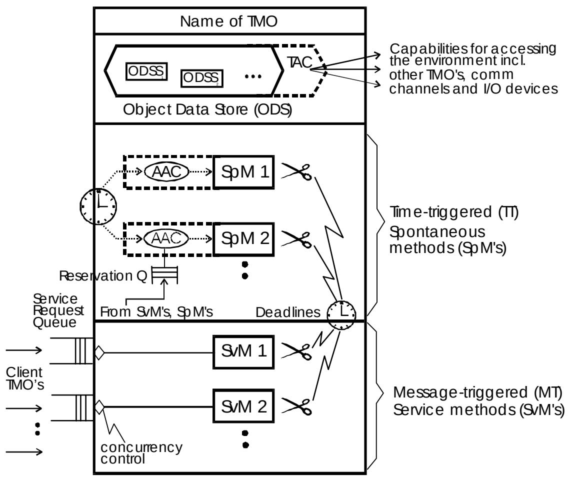
Name of TMO
ODSS
ODSS
TAC
Capabilities for accessing the environment incl. other TMO's, comm channels and I/O devices
Object Data Store (ODS)
AAC
SpM 1
AAC
SpM 2
Time-triggered (TT) Spontaneous methods (SpM's)
Reservation Q
Service Request Queue
From SvM's, SpM's
Deadlines
SvM 1
Client TMO's
SvM 2
Message-triggered (MT) Service methods (SvM's)
concurrency control

Figure 2. The TMO structure (adapted from [Kim94a])

3. Supported design and execution schemes

3.1 The TMO structuring scheme

The *Time-triggered Message-triggered Object (TMO) model*, formerly called the RTO.k object, has been developed in recent years to support rigorous and cost-effective development of real-time systems [Kim94a, Kim97b]. The TMO has been formalized to possess a concrete syntactic structure and execution semantics.

The basic structure of the TMO is shown in Figure 2. It is an extension of the conventional object model(s) in three major ways:

(a) Spontaneous method: The TMO contains a new type of methods, *time-triggered (TT-) methods*, also called the *spontaneous methods* (SpM's), which are clearly separated from the conventional *service methods* (SvM's). The SpM executions are triggered when the real-time clock reaches specific values determined at design time whereas the SvM executions are triggered by service request messages from clients. Moreover, actions to be taken at real times which can be determined at the design time can appear only in SpM's.

Triggering times for SpM's must be fully specified as constants during the design time. Those real-time constants appear in the first clause of an SpM specification called the autonomous activation condition (AAC) section. An example of an AAC is

"for t = from 10am to 10:50am every 30min
start-during (t, t+5min) finish-by t+10min"
which has the same effect as
{"start-during (10am, 10:05am) finish-by 10:10am",
"start-during (10:30am, 10:35am)
finish-by 10:40am"}.

A provision is also made for making the AAC section of an SpM contain only candidate triggering times, not actual triggering times, so that a subset of the candidate triggering times indicated in the AAC section may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an SvM within the same TMO requests future executions of a specific SpM. The AAC specifying candidate triggering times rather than actual triggering times starts with a declaration "if-demanded".

(b) Basic concurrency constraint (BCC): Under this BCC, SvM's cannot disturb the executions of SpM's and the designer's efforts in guaranteeing timely service capabilities of TMO's are greatly simplified. Basically, *activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place.* An SvM is allowed to execute only if no SpM that accesses the same portion of the object data store (ODS) to be accessed by this SvM has an execution time window that will overlap with the execution time window of this SvM. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions. We want to emphasize here that the full guarantee of the timely service capabilities of TMO's is a combination of the responsibility of the designer and the implementor and we assume this in the rest of this paper.

(c) A deadline is imposed for each output action and completion of a method of a TMO.

(d) Explicit connections to the network environment: Possible data members in the ODS include TMO access capabilities, i.e., unique ID's of the services available from other possibly remote TMO's, and programmable logical channels which are associated with both message queues, and distributed shared variables.

Extensions (a), (b), and (d) are unique to the TMO model in comparison with other object models [Att91, Ish90, Tak92]. Client methods (SpM's or SvM's) may request for service of SvM's in other TMO's. To maximize the concurrency in execution of client and server methods, client methods are allowed to make non-blocking types of service requests to SvM's.

The designer of each TMO indicates the *deadline for every output* produced by each SvM (and each SpM which may be executed on requests from SvM's) in the specification of the SvM (and some relevant SpM's) and advertises this to the designers of potential client objects. The designer of the server object thus guarantees the timely services of the object. Before determining the deadline specification, the server object designer must convince himself/herself that with the object execution engine (hardware plus operating system) available, the server object can be implemented to always execute the SvM such that the output action is performed within the deadline.

3.2 The distributed recovery block (DRB) scheme

The distributed recovery block (DRB) scheme [Kim94b] is an approach for realizing both hardware fault tolerance and software fault tolerance in real-time distributed and/or parallel computer systems. The underlying design philosophy behind the DRB scheme is that a real-time distributed/ parallel computer system can take the desirable modular form of an interconnection of *computing stations*, where a computing station refers to a processing node (hardware and software) dedicated to the execution of one or a few application tasks.

In its basic configuration, a *DRB computing station* consists of two processing nodes executing two functionally equivalent tasks, the first node called the primary node and the second node called the shadow node. Here the task software in the primary node as well as that in the shadow node are constructed by use of the recovery block language construct [Ran95]. A recovery block may consist of multiple versions of a task procedure and an acceptance test (AT) function designed to judge the reasonableness of the results produced by each version. Such versions are called try blocks. A try (i.e., execution of a try block) is always followed by an AT execution. A try not completed within the maximum execution time allowed for each try block due to hardware faults or excessive looping is also treated as an AT failure. Therefore, the AT is a combination of both *logic* and *time* AT's and can be constructed entirely in software or in the form of a software-hardware combination.

In most cases a recovery block containing just two try blocks, a *primary try block* and an *alternate try block*, is designed. The roles of the two try blocks are assigned differently in the two partner nodes. The governing rule is that *the primary node tries to execute the primary try block whenever possible whereas the shadow node tries to execute the alternate try block.* Therefore, the primary node X uses the primary try block A as the first try block initially whereas the shadow node Y uses the alternate try

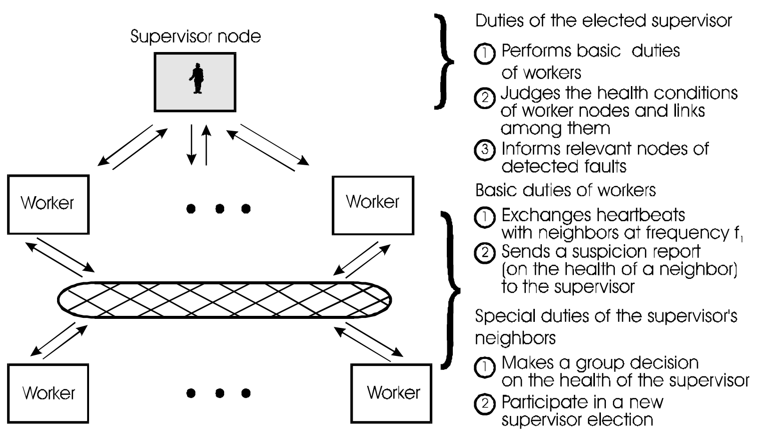
Supervisor node
Worker
Worker
Worker
Worker
Duties of the elected supervisor
1 Performs basic duties of workers
2 Judges the health conditions of worker nodes and links among them
3 Informs relevant nodes of detected faults
Basic duties of workers
1 Exchanges heartbeats with neighbors at frequency f1
2 Sends a suspicion report (on the health of a neighbor) to the supervisor
Special duties of the supervisor's neighbors
1 Makes a group decision on the health of the supervisor
2 Participate in a new supervisor election

Figure 3. The SNS scheme (adapted from [Kim97c])

block B as the initial first try block. Until a fault is detected, both nodes receive the same input data, process the data using two different try blocks, and check the results using the AT concurrently. After the primary node performs its AT execution, it informs the shadow partner of the AT result. Then, the primary node delivers the data processing results to the successor computing station(s) while the shadow node skips the corresponding output step and waits for an *output success notice* from its partner. Once the shadow receives this notice from the primary, it proceeds to enter the next execution cycle.

If the primary node fails and the shadow node passes its own AT, the shadow node learns of the failure of the primary partner either from a failure report from the partner or through a time-out for a report. Then the shadow immediately changes its role to that of the primary and delivers its processing results to the successor computing station(s). Meanwhile, the primary node, if alive, will attempt to become a useful shadow without disturbing the (new) primary node; it attempts to roll back and retry with its second try block *B* to bring its application computation state including its local database up-to-date.

3.3 The supervisor-based network surveillance (SNS) scheme

Network surveillance schemes facilitate the fast learning by each interested fault-free node in the system of fault occurrences or repair completion events occurring in other parts of the system. We recently devised a supervisor-based network surveillance (SNS) scheme [Kim97c] which is effective in point-to-point networks with a variety of topologies. Figure 3 shows the basic operation of the SNS scheme. As shown in the figure, there are two types of nodes that execute the SNS scheme, the *worker* nodes and a *supervisor* node. The worker nodes are mainly responsible for judging their own health status, the health status of their neighbor nodes, and the health status of the links attached to themselves. The supervisor node performs all the duties that a worker normally does. In addition, it is responsible for collecting *fault suspicion reports* from worker nodes, using the collected information to judge which area is the fault source, and then sending the *fault occurrence notice* to all the healthy worker nodes in the system. In case the current supervisor is judged to be faulty by the healthy neighbors, they participate in a new supervisor election and the newly elected supervisor informs all the healthy worker nodes about the fault in the old supervisor as well as the assumption of its new supervisor role.

3.4 Adaptation policies

The target applications for the ROAFTS middleware are the new generation of complex mission-critical applications (such as military planning, monitoring and command-control applications and emergency-aid hospital networks, etc.). These systems are inherently distributed and must operate in highly dynamic environments. Moreover, timing and reliability requirements of these applications may vary during their life-span. In such large-scale dynamic systems, fault-handling and resource management capabilities cannot be statically configured mainly because the amount of resources required to meet the application needs with statically configured fault-handling becomes prohibitively large.

The adaptation policy employed by the ROAFTS middleware has the following major characteristics:

(1) Improved resource utilization while retaining minimal reliability and performance requirements for critical applications: The main objective of the employed adaptation policy is to optimize system resource utilization with the constraint that reliability and performance requirements of a critical application task (which are defined based on the current environmental condition, user profile, and available resources) must be satisfied. However, in the case of resource shortage, reliability or performance requirements for less-critical tasks can be sacrificed.

(2) Rapid response to changes in the environment, system, and user profile: The adaptation manager of ROAFTS uses reports from monitoring and diagnosis components, e.g., components executing the SNS scheme. These components rapidly recognize or may even forecast any changes in system resource and/or environmental conditions and cause the adaptation

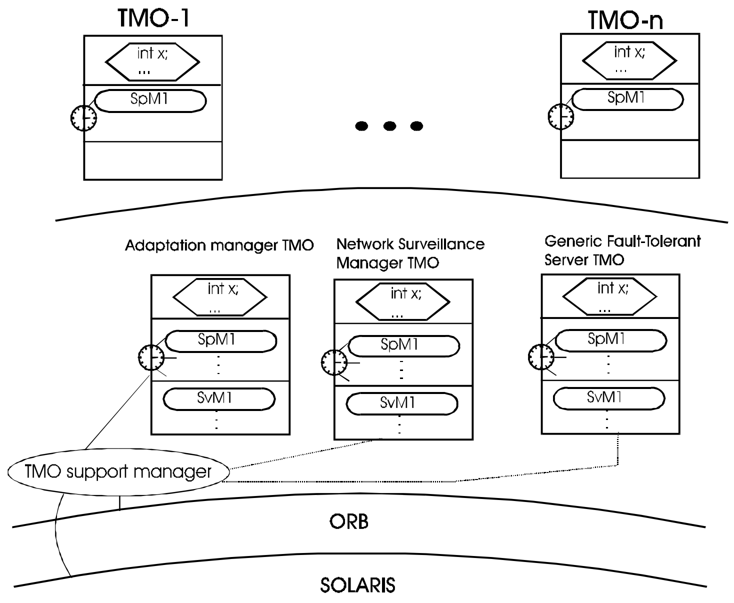

Figure 4. ROAFTS middleware implementation

manager to initiate reconfiguration decisions whenever appropriate.

(3) Reflection of application-specific adaptation parameters in the decision-making: Many of the adaptation decisions are usually application-specific. The adaptation manager in ROAFTS is thereby designed to accept application-specific parameters and reflect those parameters in the adaptation decisions.

(4) Scalability: The target applications may involve use of wide area networks. Hence, it is required that the middleware be scalable for both LAN and WAN settings. ROAFTS is based on the view of a WAN as a set of LAN's, each of which has the responsibility of managing local resources.

4. TMO-based implementation of ROAFTS middleware

In this section, major issues in the implementation of various ROAFTS middleware components will be discussed. As shown in Figure 4, ROAFTS/Solaris, built on top of the Solaris operating system and the CORBA2.0-compliant Orbix, is composed of:

- *TMO Support Manager*: This is basically responsible for supporting timely execution of TT-methods and message-triggered methods of the registered TMO's. A TMO and its methods are registered with the TMO Support Manager and the registration involves recording essential characteristics such as activation conditions for TT-methods and method completion deadlines.
- *TMO-structured Network Surveillance Manager (NSM)*: NSM is structured as a TMO and is composed of TT-methods for sending heartbeats to the neighbor nodes and discovering faulty nodes and/or links.
- *TMO-structured Generic Fault Tolerance Server (GFTS)*: GFTS is a component which provides support for different fault tolerance schemes such as the DRB scheme, the sequential recovery block scheme [Ran75, Ran95], and a forward recovery oriented exception handling scheme. GFTS has the capability to receive adaptation commands (from the adaptation manager layer or the user) to switch from one fault tolerance mode to another. GFTS provides largely user-transparent adaptable fault tolerance services.
- Due to the limitation of the current GFTS, supported applications can only be structured as simple TMO's each of which contains only one TT-method. The main reason for this restriction is that the provision of full application-transparent fault tolerance to a full-blown TMO requires additional supports [Kim97a] that have not yet

been implemented into this GFTS. The enhancement of the current GFTS implementation to support fully general active replication of TMO's is underway.

4.1 TMO Support Manager implementation

A TMO registers with the TMO Support Manager by passing the timing parameters of its methods. This registration information is used for timely activation and execution of the methods of the registered TMO. The TMO Support Manager is implemented by using the following threads:

- *TMO Manager thread*: The TMO Manager thread handles the scheduling and execution of all other threads in the ROAFTS middleware. It is implemented as a Solaris real-time thread and is periodically activated by a software watchdog timer. Since a Solaris real-time thread has the highest possible priority and is guaranteed to be executed "immediately" after its activation, the TMO Manager thread acts in a highly predictable manner. However in Solaris, an I/O handling activity cannot always be preempted promptly, and thus even the time-triggered reactivation of a real-time thread can be delayed substantially. For reasons such as this and others, the timing predictability of the application execution is still quite limited in the Solaris environment.
- *Server-Connection thread*: The Orbix on the server side requires a thread to be introduced as an interface between the server objects and the underlying communication services. The Server-Connection thread, acting as this interface, must periodically

become active to handle all incoming CORBA service requests and outgoing result messages.

- *Service-Request threads*: In order to maximize the concurrency in making service requests, we adopted the approach of creating and dedicating a thread to the chore of initiating a service-request in the client object. Whenever a client TMO method attempts to call a remote TMO (or any CORBA) method, the TMO Support Manager creates a Service-Request thread dedicated to making the remote method call. This thread will block until the result is returned from the server object. This client-side multi-threaded service request is incorporated for two reasons: (i) a TMO method can concurrently conduct multiple service-call activities and (ii) if the inter-node communication experiences a transient untimely behavior, the timing behavior of the client object remains largely unaffected.

As briefly discussed in Section 2, service methods are activated by calls from client objects. The remote activation of an object method must be intercepted by the TMO Support Manager thread (on the server side) to enforce timely activation of a relevant thread. This interference is made possible by the "filter" facility provided by Orbix. Orbix allows the application designer to provide additional code to be executed before or after the server program is executed and the code is introduced in the form of a filter [Orb95]. The filter code also enables the application designer to enforce protective measures in the form of security checks and debugging traps.

Figure 5 presents how the filter mechanism is used during the remote method calls made by client TMO methods. When the ORB in the server host receives a service request from a remote object, it first executes the filter code which is provided to (i) create a thread for executing the service request but keep the thread in the suspended mode and (ii) register the corresponding service method with the middleware. In Figure 5, this series of activities are denoted by action numbered 4. The execution of the service request will then be initiated at the appropriate time by the TMO Manager thread (action number 5 in the diagram). When the service execution is completed, the result will be forwarded back to the client object through the Server-Connection thread.

4.2 TMO-based implementation of SNS scheme

As discussed earlier, the Network Surveillance Manager (NSM) is an important component of the middleware architecture that provides fast fault detection notices to the Generic Fault Tolerance Server (GFTS). In our current implementation, the NSM has been implemented as a TMO which uses the services of the

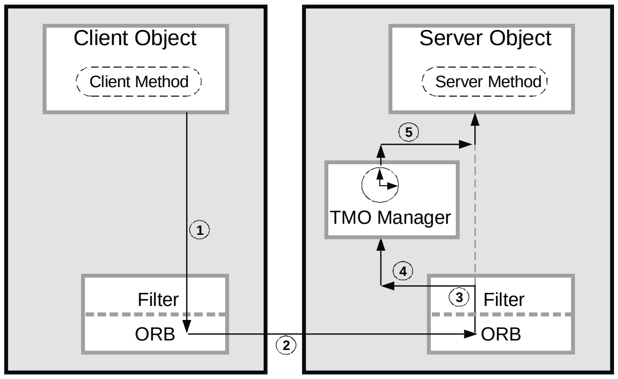

Figure 5. Anatomy of remote method calls

CORBA ORB for exchanging messages with peer NSM's executing in other nodes. This NSM currently implements the SNS scheme for point-to-point networks, even though it can be easily adapted to incorporate other complementary NS schemes. This implemented version has two periodic SpM's, called the *Heartbeat Generator* and the *Heartbeat Analyzer*. The *Heartbeat Generator* SpM is responsible for periodically generating heartbeat signals and using ORB services to send it out to the neighbor nodes. The *Heartbeat Analyzer* SpM is responsible for periodically analyzing the heartbeat signals received from the neighbor nodes. Once the supervisor node makes a judgement on a detected fault, relevant instances of the distributed GFTS are informed of detected fault occurrence. GFTS, which is also structured as a TMO, receives this notice via a shared C++ object, which is periodically checked by an SpM in the GFTS. The NSM TMO also has two SvM's, the *Heartbeat receiver* SvM and the *Heartbeat frequency modifier* SvM. The former SvM is responsible for receiving the heartbeat signals from neighbor nodes and recording them into an ODS segment (ODSS) X. The latter SvM is responsible for receiving a command from the adaptation manager regarding the changing of the heartbeat generation frequency. This SvM will register the new execution periods of the *Heartbeat Generator* SpM and the *Heartbeat Analyzer* SpM with the TMO Support Manager.

4.3 TMO-based implementation of the Generic Fault Tolerance Server (GFTS)

In this section, we will first discuss the way application-transparent fault tolerance provisions were realized. Then the TMO structure of GFTS will be presented.

It is undoubtedly desirable to separate fault management concerns from the concerns for the functionalities of the applications. However, there are some fault management elements which are application-specific. For example, a non-trivial and realistic acceptance test (which checks the reasonableness of application states or outputs) varies from one application to another. Therefore, it is useful to isolate application-independent elements from application-specific ones. Using object-oriented design techniques, it is feasible to capture the application-independent features as a base class such that each application can seamlessly inherit these features and add application-specific features in application-dependent derived classes. In the ROAFTS middleware the GFTS provides fault tolerance services as a set of abstract classes from which the customized objects can be constructed.

GFTS is structured as a TMO with two SpM's and three SvM's. The *GFT Executive* SpM is responsible for executing the associated application using the selected fault tolerance scheme (which is given by either the user or an external adaptation manager). The selected fault tolerance scheme can be one of the following: (i) the Distributed Recovery Block (DRB) scheme, (ii) the Sequential Recovery Block scheme, or (iii) a forward recovery oriented Exception Handling scheme. The *External Command Interpreter* SpM periodically checks an external C++ object for a command from an external adaptation manager for changing the current fault tolerance mode or for a command from the NSM for any fault detection notice. If it finds such a command, it notifies the *GFT Executive* SpM via a signal deposited in an ODSS so that the latter may take an appropriate action.

The Input Coordinator SvM is responsible for receiving the input sent from the predecessor TMO(s) and queuing the input in an ODSS. The queued input data items will be processed by the *GFT Executive* SpM which will execute a try block. In schemes such as the DRB scheme in which different versions of an application task execute in two or more nodes, it is imperative for the primary node to send status messages such as data ID, acceptance test result, etc., to the shadow node. The *Peer Status Receiver* SvM in a shadow node is responsible for receiving the status messages sent by the primary partner's *GFT Executive* SpM. Thus, GFTS in each primary node sends status messages for its partner GFTS by making a remote method call to the partner's *Peer Status Receiver* SvM. Finally, in order to facilitate orderly joining of a new shadow partner, a newly joined GFTS requests its active partner to send the most recent application state, by invoking the *Partner Rejoin Request Receiver* SvM of the primary GFTS.

4.4 TMO-based implementation of the Adaptation Manager

The Adaptation Manager is structured as a TMO with one SpM and three SvM's. The *Adaptation decision maker* SpM is periodically to investigate the current system and environment state using the *Environment State* TMO and the *Internal State* TMO and then to decide whether any adaptation must be done. In case an adaptation decision is taken, it will inform the relevant GFTS and/or the NSM instances of the decision and also instruct the *Resource allocation executive* TMO to reallocate system resources based on the current system state.

The adaptation decision can also be activated when the NSM TMO sends a notice requiring fast adaptation. The *Recv notification* SvM is responsible for receiving such a notice from NSM and making a reservation for an activation of the *Adaptation decision-maker* SpM at the earliest predesiged activation time. The user can dynamically change the reliability and/or performance requirements of exisiting applications via a GUI by calling the *Recv application requirement change* SvM. If this SvM recognizes a significant change in the requirements of a critical application, then make a reservation for an activation of the *Adaptation decision-maker* SpM at the earliest time that can be reserved.

If the user feels that the automatic adaptation carried out by ROAFTS is not sufficient for satisfying the application requirements, it can order the adaptation manager to temporarily abandon automatic adaptation decision-making and then act merely as an agent that carries out the wishes of the user.

5. Our experiences with the ROAFTS/Solaris

Our experiences have shown that ROAFTS/Solaris can support real-time applications with time granularity in the range of 40 - 100 msec well if the local area network executing the middleware is isolated from the outside world (e.g., Internet). The time-slice for the thread scheduling is currently chosen to be 100 msec to satisfy the needs for target applications such as military distributed planning. Although the temporal predictability of ROAFTS/Solaris is weaker than those of the implementations which are conducted in less open environments such as isolated intra-nets equipped with more specialized operating systems, we believe ROAFTS/Solaris can still be effectively used in a wide range of moderate-precision real-time applications.

The ROAFTS/Solaris implementation experiences provided us insights into two major issues, (i) shortcomings and potentials of the Solaris operating system in a real-time setting, and (ii) effectiveness of the current CORBA standard for use in real-time applications. We believe that system designers must consider these issues carefully in order to effectively utilize COTS operating systems and CORBA middleware for real-time applications.

As mentioned earlier, Solaris has incorporated multi-threading and real-time thread capabilities in addition to traditional Unix in order to extend Unix-like operating systems for use in real-time applications. While being effective in soft real-time applications, Solaris shows several limitations in this area. One major shortcoming is that when a real-time thread issues a "slow" system call (a system call which causes the caller to be blocked until the requested service is completed such as pipe inter-process communication, socket-based communication, and IO services), the execution resources are shared by other non-real-time processes and/or threads during the execution of the system call. In other words, if an application running in the real-time mode issues system calls, its timing behavior cannot be well predicted from then on.

Another lesson learned from the ROAFTS/Solaris implementation is related to the applicability of the CORBA-compliant ORB's for real-time applications. Although CORBA provides a clean high-level abstraction for location-transparent language-independent inter-object communications, it must be extended in the future to meet the following two requirements:

- A majority of current implementations are built on top of non-real-time operating systems and non-real-time communication protocols. If the underlying operating system and communication subsystem cannot guarantee predictable services, the CORBA middleware will not be able to guarantee timely interaction among remote objects as well as local objects.
- Even if CORBA is built on top of real-time operating systems, it will not be able to guarantee timely interacton among objects, mainly because the application designer does not have the means of defining timing characteristics of the applications such as the completion deadline for service-requests (method calls).

Object Management Group has recognized the need to define a new standard real-time CORBA for real-time applications [OMG96]. This work is at the very early stage. It seems worth noting here that the real-time application domain ranges over a wide spectrum of applications with different degrees of predictability requirements and thus this fact should be reflected in defining a real-time CORBA standard.

Acknowledgments: The research work reported here was supported in part by the US Defense Advanced Research Project Agency under Contract N66001-97-C-8516 monitored by NRaD, in part by the University of California's MICRO Program under Grant 96-169, in part by Hitachi, Ltd., in part by LG Electronics, and in part by the USAF Rome Lab under Contract F30602-96-C-0060.

5. References

[Att91] Attoui, A. "An Object Oriented Model for Parallel and Reactive Systems", *Proc. IEEE CS 12th Real-Time Systems Symp*., 1991, pp. 84-93.

[Hur96] Hurley, P., and Dussault, J. L., "The Development and Assessment of An Adaptive Fault Resistant System (AFRS)", *Proc. Second International Command & Control Research & Technology Symposium*, Warwickshire, UK, Sept., 1996, pp. 256-267.

[Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.

[Kim92] Kim, K. H., and Lawrence, T., "Adaptive Fault-Tolerance in Complex Real-Time Distributed Applications", *Computer Communication*, Vol. 15, No. 4, May 1992, pp. 243-251.

[Kim94a] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", *Proc. 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Oct. 94, Dana Point, pp.36-45.

[Kim94b] Kim, K.H., "Action-level Fault Tolerance", Ch. 17 in Sang H. Son ed., *'Advances in Real-Time Systems'*, Prentice Hall, 1994, pp. 415-434.

[Kim97a] Kim, K. H., and Subbaraman, C., "Fault-Tolerant Real-Time Objects", *Communications of ACM*, Vol.40, No. 1, Jan. 1997, pp. 75-82.

[Kim97b] Kim, K. H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No. 8, August 1997, pp. 62-70.

[Kim97c] Kim, K.H., and Subbaraman, C., "A Supervisor-Based Semi-Centralized Network Surveillance Scheme and the Fault Detection Latency Bound", to appear in *Proc. 16th Symposium on Reliable Distributed Systems*, October 1997, pp. 146-155.

[OMG95] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, 1995.

[OMG96] OMG Real-Time Interest Group, "Real-Time CORBA Issue 1.0", Working Document, OMG, 1996.

[Orb95] *Orbix Programming Guide*, IONA Technologies, Dublin, 1995.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance.", *IEEE Transactions on Software Engineering*, June 1975, pp.220-232.

[Ran95] Randell, B., "The Evolution of the Recovery Block Concept", Chap. 2 in *'Software Fault Tolerance'*, Michael R. Lyu, Editor, 1995, pp. 1-21.

[Sho97] Shokri, E., et al., "An Approach for Adaptive Fault Tolerance in Object-Oriented Open Distributed Systems", *Proc. 3rd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 97)*, Newport Beach, CA, February 1997, pp. 89-98.

[Tak92] Takashio, K., and Tokoro, M., "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", *Proc. OOPSLA*, 1992, pp. 276-29.