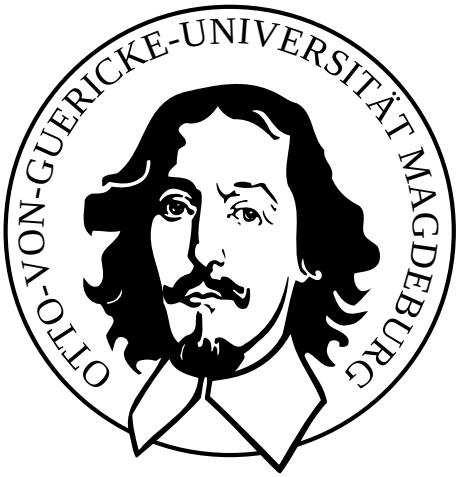
Multi threading in a CORBA ORB

– Diplomarbeit –

Andreas Schultz
<aschultz@cs.uni-magdeburg.de>

8. Januar 2001



„Otto-von-Guericke“ Universität Magdeburg
Fakultät für Informatik
Institut für Verteilte Systeme

Aufgabenstellung:	Prof. Dr. Wolfgang Schröder-Preikschat
Betreuung:	Dr. Arno Puder – Deutsche Telekom San Fransico
	Andy Kersting – Sr. Software Engr. AT&T Broadband Denver

Contents

1 Introduction and Motivation

Current work on CORBA implementations with multi-threading support is either focusing on highly optimized multi threaded ORB's (mt-ORB) for SMP platforms or on real-time-mt-ORB's with strict Quality of Service (QoS) demands for embedded or special purpose systems. This has lead to a number of highly specialized systems following their own, sometimes non CORBA compliant, strategies. [1][2]

Common to all approaches is the attempt to modify existing ORB structures to include multi threading capabilities in the hope to create a high-speed multi-thread ORB. The goal of this work was not to modify existing structures, but to provide a highly scalable, robust and flexible *internal* framework for multi threaded ORB's to support a broad range of threading, scheduling and Quality of Service (QoS) models including real-time-prioritization of threads and messages. Special care should be taken to make it possible to support external hardware devices or specialized I/O processors as ORB components in this framework.

The work for this thesis consisted of four major parts:

Analysis:
To understand the requirements of a mt-ORB it is necessary to understand the requirements of mt-applications using such an ORB. Currently used mt-ORB models have to been investigated and evaluated under typical usage conditions. *It was not the goal of this work to find the best threading or scheduling model for an mt-ORB.* It will be in the responsibility of the application architect to choose from a number of possible configurations the appropriate one.

Design:
There has been extensive work done by the OMG [3] to define a real-time extension to CORBA. This extension focuses exclusively on the ORB-API. While real-time is not equivalent to multi-threading, have both to deal with the same problems at the API level.

The design was therefore focused on the thread-to-scheduler interface, the thread-to-thread communication interface and the inter-thread synchronization interfaces. It is important that those components fit within the framework and fulfill the requirements laid out in the real-time CORBA specification. They are to provide the basic building blocks to later implement those real-time facilities.

Application to MICO:
In this step the framework defined during the design has been applied to MICO. This included the identification and definition of atomic threadable operations within the MICO-ORB.

Implementation:

This final step resulted as proof of concept in a prototype implementation of a MICO mt-ORB for a standard UNIX platform.

1.1 Introduction to CORBA

The Common Request Broker Architecture (CORBA) by the Object Management Group (OMG) was first published in 1991. CORBA is an object oriented programming language, hardware and operating systems independent communication middle-ware. It is a vendor neutral *specification*.

Other middle-ware platforms in use today besides CORBA are Remote Procedure Calls (RPC)[1], the Distributed Component Object Model (DCOM) and Java's Remote Method Invocations (RMI).

The OMG positions CORBA as an universal middle-ware communication platform to solve all application communication needs. Their vision is that in the future the communication with all types of appliances and applications uses CORBA allowing universal unrestricted interaction between them. Currently work is underway to extend the reach and usability of CORBA into the realm of real time communication and embedded systems. Todays available implementation are huge pieces of software often with a tremendous overhead of functions not need for those areas. The upcoming CORBA 3 standard will introduce definitions for minimum CORBA (a small and efficient subset of CORBA targeted at embedded systems) and real time CORBA (which will define interfaces to express quality of service restraints and support real time demands).

The OMG attempts to position CORBA in the embedded and real time domain are not without critiques. Many experts doubt the ability of CORBA to meat the requirements of those systems. Embedded systems are typically low end microprocessors with very limited memory, usually between 4k Byte to 32k Byte. Todays CORBA implementations, even when reduced to the interfaces defined in minimum CORBA, are of some orders larger than that. The sophisticated mechanisms used by CORBA to encode, marshal and demarshal invocation requests and method parameters add additional overhead to method invocation which can make it difficult to meat even soft real time constraints.

The OMG states in [4]:

> The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them.

That certainly is a complex goal. The CORBA standard defines four interface to archive that goal:

1 RPC means here not the concept of remote procedure calls but the concrete implementations of those as in Sun RPC and DCE RPC

Interface Description Language (IDL)
Object Interfaces are describe in every programming language differently. The IDL provides uniform, programming language independent means to describe object interfaces. IDL descriptions are translated to the target implementation language by the IDL compiler. The concrete code generated depends on the target system and language, but is always transparent in it functionality to the application code.

Object Request Broker (ORB) – API
Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems. The ORB interface is defined using the IDL and is guaranteed to be consistent across all CORBA compliant ORB's.

Object Adapter (OA)
Object adapters are the mediators between the ORB and the application server objects. They are responsible for delivering invocation requests to the objects and maintain state and control information about the objects.

General Inter-ORB Protocol (GIOP)
GIOP defines the communication protocol used to link ORB's together. It can be mapped onto different network protocols like TCP/IP or IPX. It sets the rules for request and parameter encoding and ensures that data can be read and interpreted regardless of hardware specific encodings.

The ORB-API and the IDL generated code form a horizontal interface between the ORB and the application. A vertical interface between ORB's is defined by the GIOP (↗ Figure 1.1).

A CORBA system can be divided in three abstract layers (↗ Figure 1.1) which can be mapped onto the ISO/OSI reference model (↗ [5]). The application layer consists of the code the application developer writes and the code generated by the IDL compiler. This layer would is equivalent to the application layer in the ISO/OSI reference model. The ORB layer consists of the complete ORB and support code. In the ISO/OSI reference model it would provide layer 6 (presentation layer) and part of layer 5 (session layer) services. The communication layer consists of services provided by the operating system or third party libraries and provides anything below and part of layer 5 in the ISO/OSI reference model. Currently the CORBA standard defines only a TCP/IP socket based communication model.

The CORBA standard requires that the used communication mechanism is at least connection oriented and reliable detects error conditions. It then defines its own additional connection handling and error recovery mechanisms on top of that. A CORBA connection can survive the disconnection of the physical connection it is based on and can transparently reestablish contact to the remote side when needed.

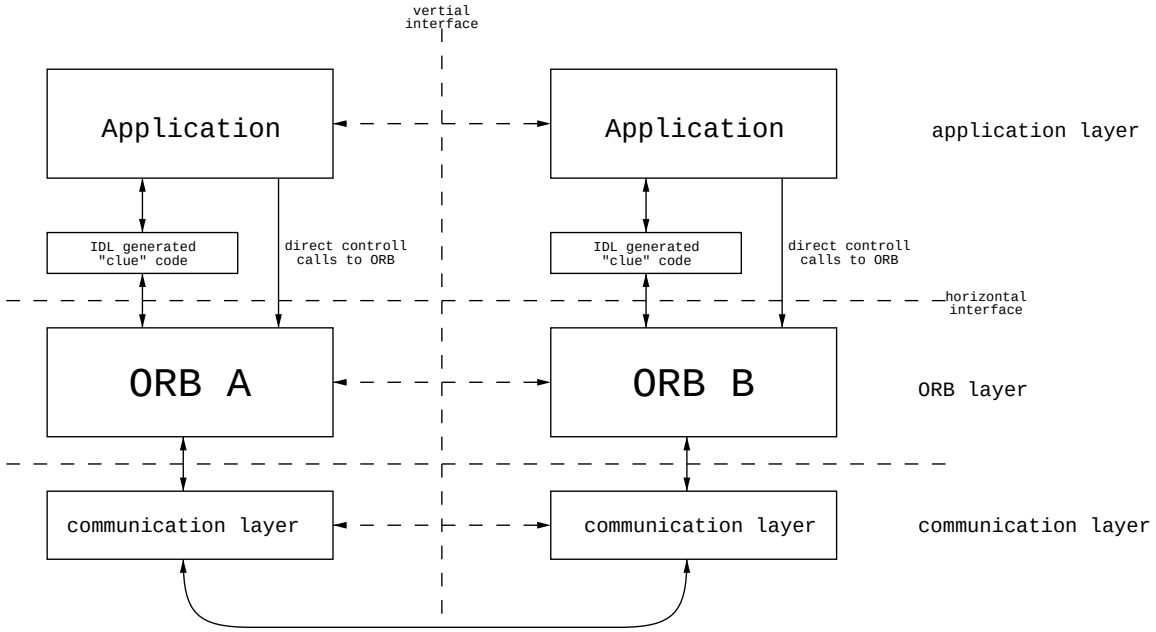


Figure 1.1: General structure of CORBA based applications

2 The use of threads in CORBA applications and implementation

Threads can significantly improve an applications structure and can also help making its development more intuitive by delegating specific tasks to threads which otherwise would have required sophisticated mechanisms to integrate them into a single execution flow. Popular example for thread usage are asynchronous blocking I/O, possibly long running or blocking code sequences like database queries, sorting of large amounts of data or number crunching. Multi threading also allows to make full use of the CPU resources provided in multi-processor systems.

While an ORB can be used to invoke methods on objects in the same application context, is the most common case invoking methods on an ORB in a different application context whether on the same system or on a remote system.

By doing so CORBA applications exploiting already one level of parallelism. Adding a second level of parallel execution to this by using threads on the client or server side creates a whole new set of problems. Some of those problems have to be addressed by the CORBA implementation itself while others are the sole responsibility of the application developer. It is important to note that not the use of threads in the application itself has consequences for the ORB. Only when multiple threads use methods provided by the CORBA implementation has the ORB to deal threading related problems.

The following sections first discuss the different types of CORBA implementation in respect to their multi threading capabilities, an then gives an introduction for their use in certain application scenarios.

2.1 Types of CORBA implementations

CORBA implementations can be classified by their amount of thread awareness and use into:

- thread unaware
- multi threaded
- thread aware but single threaded

2.1.1 Thread unaware CORBA implementations

A thread unaware CORBA implementation knows nothing about threads and therefore does not employ synchronization mechanism to protect shared data structures. Only one thread may

execute code belonging to the CORBA implementation at any given time. The application programmer is responsible for guaranteeing that by any means applicable.

The lake of parallelism leads to straightforwared code that needs no special consideration to protect data integrity or synchronize threads within the ORB.

The lack of thread support also has its impacts on the functionality an ORB is able to provide. Without threading, I/O operations have to be performed in a polling manner and the ability to provide asynchronous mechanisms is very limited. Furthermore any operations that blocks, and certain I/O operations can block even when used in a nonblocking fashion, will completely stall the ORB until its return.

2.1.2 Multi threaded CORBA implementations

Multi threaded CORBA implementations internally use threads for various task and the ORB and object adapter interfaces are fully reentrant. Several mechanisms to give user code control over various aspects of ORB internal thread usage are prescribed by the CORBA 2.2 standard and more are part of the upcoming CORBA 3.0 standard.

The ORB is able to employ blocking operations without impacting the overall application performance or responsiveness. Automatic or manual adaption to varying load demands is possible by adapting the thread usage strategies. On SMP systems and operation systems that offer kernel threads, these ORB's scale much better through the efficient use of all available CPU's.

2.1.3 Thread aware, single threaded CORBA implementations

In a thread aware CORBA implementation the ORB and object adapter interfaces are fully reentrant but the implementation itself does not actively use threads. Pure CORBA based server applications can not benefit from this architecture since incoming invocations are still being serialized by the lack of threading in the ORB itself.

The design of thread aware ORB can be simpler than a full multi threaded ORB since no mechanisms for thread and thread pool management are needed in the ORB.

2.2 End-to-End synchronization requirements

End-to-End arguments have been introduced in [6]. It refers to the fact that some mechanisms in End-to-End communication systems are better implemented in higher layers than in basic communication layers. End-to-End synchronization applies at two different places in a CORBA system. Synchronization between client and server is not covered by the CORBA specification and has to be realized if needed in the application context itself. The other place where synchronization might be required is the API between application and ORB. Thread aware and multi threaded ORB's use thread safe algorithms to manipulate internal data structures or employ appropriate synchronization mechanisms to protect them, so that applications can use the ORB without additional precautions. When thread unaware or only partly thread safe ORB's are used the application developer has the choice which level of synchronization

and protection should be employed and can adopt a scheme that is most appropriate for his purposes. On the other hand can completely thread safe and multi threaded ORB be used by applications without special knowledge or precautions. The decision which combination is the best has to be determined in each case individually.

In some cases the application context and the synchronization between client and server ensures already the proper synchronization at the ORB–application interface. Relying on these dependencies is not explicitly forbidden by the CORBA standard but does violate the spirit of CORBA which tries to provided consistent interfaces in the respect that the server side interface behavior does not change when the application context or the client type changes. This leads the requirement that CORBA based clients and servers should make no assumptions whatsoever about the type and behavior of their respective counterpart.

The following section try to give some guidelines for choosing the appropriate ORB type for certain application types.

2.3 CORBA based clients

This section deals with application that contain only client components and do not provide CORBA based server functionality.

2.3.1 Single threaded clients

In a single threaded client only *one* thread is used by the application.

The existence of only a single execution thread does not mean that the application has no need for concurrent execution of functions. The client has always to wait for the result of an invocation that contains *out* or *inout* parameters. A one-way invocation however could be handed off to the ORB and delivery to the server could occur transparently parallel to the main execution flow. The ORB is completely responsible for allocating new threads for parallel execution and their use is completely transparent to the application (↗ Figure 2.1).

2.3.2 Multi threaded clients

In a multi threaded client multiple threads are present, for the ORB however it is only significant how many threads at once try to access it.

It is relatively easy to construct a sample case in which it is sensitive to use multiple threads in an application but still only a specific thread or one thread at a time use the ORB, that means access to the ORB is serialized by means implicit to the application. That in turn means that the ORB can be thread unaware and no additional synchronization is required. Even the fact that multiple thread might try to access the ORB at the same time does not necessarily requires an thread aware ORB. Synchronization and serialization might better be performed by the application itself to meat its special requirements.

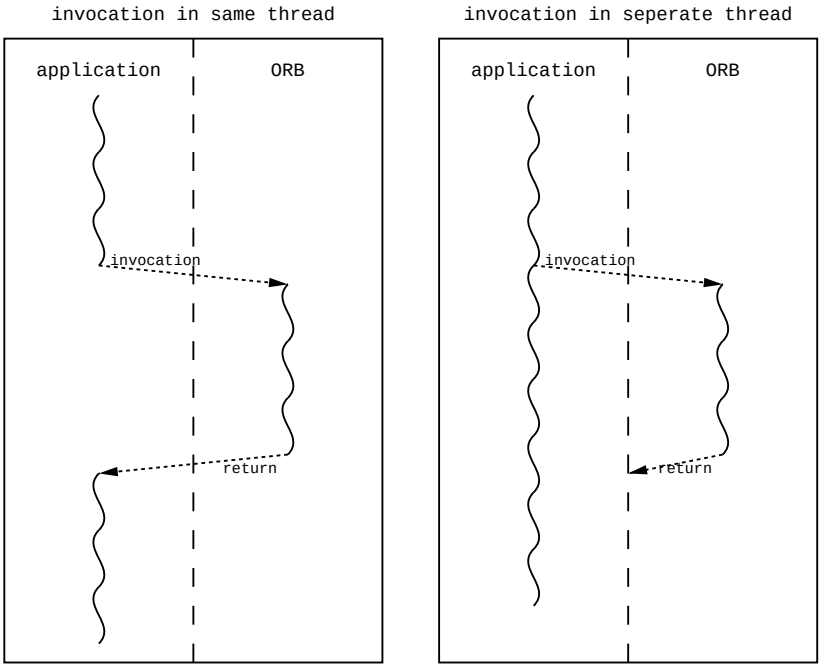


Figure 2.1: non-concurrent and concurrent invocation

2.4 CORBA based servers

The thread characteristics of applications that provide only server components are completely determined by the thread characteristics of the used ORB.

2.4.1 Single threaded ORB's

Single threaded ORB's no matter whether they are thread safe or not always serialize incoming request. The number of active concurrent requests is limited to one. In applications without client part, the object implementations do not need to be thread safe or even thread aware since all upcalls to them are already serialized.

Interlaced upcalls

Upcalls to method implementation can be interlace in special situations, that means the same method can be invoked more than once before its return. Depending on the structure of the ORB the return ordering might be important, i.e. return from the invoked method itself but also possibly from subsequently invoked methods has to occur in last-in-first-out (LIFO) order. Situations like this can only occur when control is voluntary transfered back to the ORB during an invocation. Consider the following example:

During the time a client ORB waits for the server to respond, it usually enters an event wait loop. In this loop it can respond to and process incoming requests. If in a server a method A that has been invoked by a remote client, uses a method B invoked via CORBA mechanisms,

then it is possible that while waiting for the second server to respond to the request for B a new request for the method A is received and method A is actually invoked. This type of interaction is actually needed to allow recursive invocations across more then one ORB.

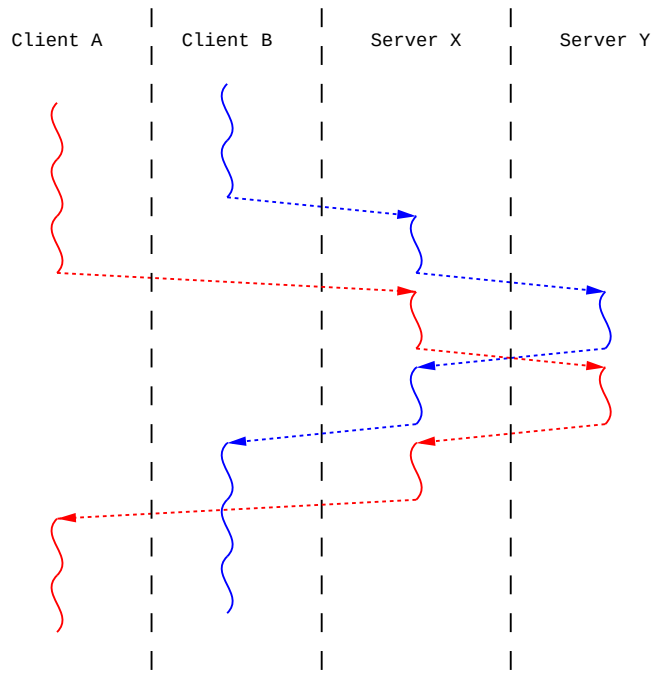


Figure 2.2: Interlaced invocation

Figure 2.2 illustrates this situation. Two clients A (red) and B (blue) access a method X on a server which in turn invokes a method Y on a second server. While X is waiting for the result from Y invokes the second client the method X. This leads to method X being active two times without the need for multi threading.

Three solutions for safely dealing with those situation are feasible. Incoming requests for already active methods can be queued in the ORB or they can be queued at the applications object level, or the application code can be written to be safe to be used with those interlaced invocation. Which solution is appropriate for a given application depends on the usage scenario, the used ORB and on the used algorithms.

2.4.2 Multi threaded ORB's

In a multi threaded ORB upcalls can be performed concurrently. Application code can ensure serialization when needed by setting the right thread policies or implement synchronization mechanisms it self. Thread unsafe code could be used without synchronization when external conditions ensure that no concurrent requests can occur. It has however to be noted that such an approach which might satisfy certain speed and complexity considerations for a specific usage scenario, does violate the intentions of CORBA to provide a uniform and *consistent* interface to server side objects no matter what the usage current scenario might be.

2.5 Interoperability of different client and server types

Generally, all types of CORBA clients can use any type of CORBA server transparently (i.e. the client has no and need no knowledge about the particular server implementation). However not all combinations can make use of advanced features like AMI, realize complicated callback structures or archive the highest possible performance.

Single threaded servers can serve only one request at a time. Throwing a multitude of concurrent request at such a server will achieve nothing. Client requests will effectively be serialized. Despite those considerations become single threaded servers in an environment with multi threaded clients or huge numbers of single threaded clients not automaticly a performance bottleneck. These servers can yield good performance results when the requests are already synchronized by the overall application context, only relatively small numbers of invocations occur or the number of concurrent requests low is. In execution environments that do not meet these criteria the use of full multi threaded servers should be considered.

Multi threaded servers can typically serve a huge number of client requests concurrently. Using them in conjunction with a single single threaded client or with multiple clients and very infrequent requests will provide no significant advantages over single threaded servers for typical applications. Applications of this profile with specific demands might still be able to benefit from this combination. Consider a server with a least two clients where client A makes periodic short requests every second and client B makes very infrequent requests but long running requests. The server in this case could possibly use a single threaded ORB without much problems except occasional delays in request answers for client A. If the application case however demands that requests of client A are not delayed longer that a certain amount of time which is shorter than the worst case time a request from client B takes to process, then either a multi threaded solution or mechanisms to interrupt and suspend request from client B are needed. Meeting the timeout criteria for client A can be realized with a multi threaded ORB which has support for QoS restraints or with a simple multi threaded ORB when external conditions (MP system with enough idle CPU's, time sharing operating system with enough idle time left, etc.) can provide the same guarantees.

3 The CORBA specification and threads

This chapter gives an introduction to the interfaces defined in the CORBA 2.3 specification which are important for multi threaded ORB's and have to be supported properly. It also provides a look at in–progress OMG specification that describe further extension that need or benefit from multi threaded ORB's and interfaces that provide additional capabilities in multi threaded ORB's.

It is assumed that the reader is familiar with CORBA and certain details of the ORB interface. For a introduction to CORBA have a look at [4].

3.1 Overview of CORBA 2.3

The current CORBA specification prescribes only a few interfaces dedicated to support for multi threaded ORB's.

The first object adapter introduced was the Basic Object Adapter (BOA). The CORBA standard gives no definition for concurrent invocations using a BOA. The behavior of this object adapter in multi threaded environments is consequently implementation dependent and non portable between different CORBA implementations.

Newer versions of the CORBA standard define the Portable Object Adaptor (POA). A POA knows about a thread policy which can have the following values:

SINGLE_THREAD_MODEL serializes all invocations on the POA. In a multi-thread environment, all upcalls made by the POA to implementation code are made in a manner that is safe for code that is multi thread unaware.

ORB_CTRL_MODEL allows the ORB and POA to choose the best method to assign requests to threads. It does not require the ORB or the POA to use multiple threads, it merely permits the delivery of concurrent request using multiple threads. In this model the application developer is responsible for protecting thread sensitive code.

Access to information associated with the current thread of execution can be obtain via the *CORBA::Current* object. Each CORBA module that wishes to provide such information does so by using an interface that is derived from the *Current* interface defined in the CORBA module. Operations on interfaces derived from *Current* access state associated with the thread in which they are invoked, not state associated with the thread from which *Current* was obtained. This prevents one thread from manipulation another threads state, and avoids the need to obtain and narrow a new *Current* in each method's thread context.

The *PortableServer::Current* interface is derived from the *CORBA::Current* interface and has the same thread context semantics as the *ORB::Current* interface. It provides method implementations with access to the identity of the object on which the method was invoked (↗ [7] sect. 11.3.9).

The current CORBA specification describes no mechanisms to control thread parameters nor does it make any prescriptions about synchronization mechanisms. Consequently all currently known multi threaded CORBA implementations deliver their own proprietary interfaces for those task. The real-time CORBA proposal, which will most likely be part of the upcoming CORBA 3.0 standard, defines standard interfaces for CORBA in real-time environments. The real-time CORBA proposal being almost exclusively aimed at conventional real-time applications provides extensive support to control the priorities of single requests though the priorities of the associated threads and connections. It has no support to express dead lines or answer latencies.

Thread related interface described in the real-time CORBA proposal are interfaces to control thread pools and a single synchronization primitive, the mutex. It does not provided the definitions necessary for a uniform and portable thread creation and control interface.

3.2 Status of relevant in progress OMG specifications

The upcoming CORBA 3 specification will contain a number of components that have direct impacts for multi-threaded ORB's and others that can greatly benefit from multi threading abilities in ORB's.

The specifications included in the designated CORBA 3 standard divide neatly into three major categories:

- Internet integration
- Quality of service control
- The "CORBAcomponent" architecture

The quality of service control category contains the important to multi threading related specifications:

- asynchronous messaging and quality of service control
- minimum, fault-tolerant, and real-time CORBA

3.2.1 Asynchronous Messaging and Quality of Service Control

The new messaging specification defines a number of asynchronous and time-independent invocation modes for CORBA, and allows both static and dynamic invocations to use every mode. Results of asynchronous invocations may be retrieved by either polling or callback, with the choice made by the form used by the client in the original invocation. Policies allow control of quality of service of invocations. Clients and objects may control ordering (by time, priority, or deadline); set priority, deadlines, and time-to-live; set a start time and end time for time-sensitive invocations, and control routing policy and network routing hop count.

3.2.2 Minimum, Fault-Tolerant, and Real-Time CORBA

Minimum CORBA is primarily intended for embedded systems. Embedded systems, once they are finalized and burned into chips for production, are fixed, and their interactions with the outside network are predictable – they have no need for the dynamic aspects of CORBA, such as the Dynamic Invocation Interface (DII) or the Interface Repository (IR) that supports it, which are therefore not included in Minimum CORBA.

Real-time CORBA standardizes resource control – threads, protocols, connections, and so on – using priority models to achieve predictable behavior for both hard and statistical real time environments. Dynamic scheduling, not a part of the current specification, is being added via a separate Request For Participation (RFP).

Fault-tolerance for CORBA is being addressed by an RFP, also in process, for a standard based on entity redundancy, and fault management control.

4 Targeted software and hardware environments

This chapter analyses the typical hardware, software and development environments ORB's are used in and concludes requirements for multi threaded ORB's from those.

4.1 Hardware challenges

CORBA is not targeted at a specific hardware platform. ORB's can therefore be implemented on all kinds of different architectures ranging from embedded platforms over PCs to mainframes. Modern operating systems provide a uniform interface to the hardware and hide most of its complexities. A lot of hardware characteristics still influence the performance of the user application. In order to archive maximum performance, ORB's have to be optimized for the specific parameters of the platform they are running on.

In the scope of this thesis the most common parameters to deal with are:

- number and distribution of CPU's
- cache organization
- memory organization

Number and distribution of CPU's

Common CPU configurations are:

- uniprocessor (UP) system
- symmetric multiprocessor (SMP) system
- massively parallel processor (MPP)
- no-uniform memory access (NUMA) systems
- clustered UP system
- clustered SMP system

Uniprocessor (UP) system

A uniprocessor system has only one CPU. True parallel execution is not possible. When using a multi-task operation system on such hardware, parallel execution is simulated by dividing CPU time in slices or slots and assigning those slices according to a central scheduling algorithm

to all active tasks. The running task can be interrupted (preempted) only after the currently active machine instruction has been completely executed. Synchronous concurrent access to the same memory location is therefore no possible, but a sequence on instructions modifying a memory location can be interrupted and need therefore to be protected if concurrent access can occur.

All operating systems provide at least basic synchronization mechanisms. These use in most case system calls to perform the functions and usually take considerable time. Simple atomic operations like increment or decrement–and–test already exists on most CPU architectures in an UP configuration and are many times faster than using system provided synchronization around non atomic versions.

Symmetric multiprocessor (SMP) system

SMP systems consist of two or more CPU's sharing the same working memory and the same bus system to access that memory. Access to the same memory location can happen simultaneously and concurrent execution of code modifying the same memory location is more likely than on an UP system. Appropriate algorithms that can work on shared data structures without corrupting them have to be used, or the access to those data structures has to be serialized. Serialized access to shared data can also lead to unintended synchronization of threads.

When using spinlocks on SMP systems for serialization can the heavy use of those locks to protect data easily lead to lock contention, a situation where a lot of threads tries to access the same data structure and has to wait to acquire the lock protecting the data.

MPP, NUMA and clustered systems

In an MPP system, each CPU (node) contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect.

A cluster uses two or more computer systems (nodes) that work together and are connected via a high-speed interconnect. Those connections are of some order slower that those used in MPP systems.

NUMA is a multiprocessing architecture in which memory is separated into close and distant memory banks, memory on the same processor board (node) as the CPU (local memory) is accessed faster than memory on other processor boards (shared memory).

Clustered UP and SMP, and MPP systems share a lot of characteristics as far as the application is concerned. Multiple code streams are executed in parallel and access to the same memory location can happen simultaneously. Data integrity issues are the same a for SMP systems. Access to memory not local to the current node is typically slower that access to node local memory and should therefore be avoided if possible.

Cache organization

Cache is usually completely transparent and is typically of no concern to the application. The memory layout and usage pattern of a program can however have huge impacts on the cache efficiency and might therefore be of interest to some applications.

Best results are usually achieved by placing thread local data structures closely together and by avoiding putting data structures that are used in different threads in the same cache line. Normal memory allocators have no knowledge about the applications memory usage patters. The use of application specific memory allocators and additional developer provided hints can result in dramatic speed improvements (↗ [8]).

Conclusions

Multi-threaded ORB's should keep thread local data in memory locations that are private and local to the thread. Access to shared data structures has either be to be synchronized or the algorithms used have to be able to deal with concurrent access to those data. Access to data residing in non local and therefore potentially slower memory should be minimized.

4.2 Software challenges

The ORB is the central control instance for thread usage throughout the CORBA implementation. There are a number of questions and problems that the ORB has to address.

Thread lifetime

Threads are created, perform a certain amount of work and are finally terminated. The decision when to create new threads and when to terminate them depends upon the application structure, resource availability and the used thread package.

Application structure and resource availability are parameters that are outside the scope and control of the ORB. Only the application designer has informations about the specific demands of his product, furthermore those parameters change dynamically at runtime. The ORB has to be adaptable to these demands. This requires the ability to change a wide range of parameters at runtime.

Thread creation

Depending the kind of thread package used, the operating system characteristics, the available resources and the applications needs can threads be created on the fly as needed or preallocated and reused.

Thread preallocation and re-usage typically requires a sophisticated management framework. Additionally, on some thread packages, thread creation can be faster than reactivation of blocked or sleeping threads. The complexity of the used management framework and the thread activation policy has to be determined at ORB compile-time, since it heavily depends on target system specifics.

Thread destruction

Thread termination usually includes the release of associated resources. It can occur as a normal thread return or through asynchronous or synchronous cancellation.

Cancellation requires the ability to identify all resources associated which the thread, in order to be able to release those in an orderly fashion. Failure do to so can result in inconsistent states of synchronization primitives and resources leaking.

Invocation modes

Several schemes for mapping invocations to threads exist for multi threaded ORB's. Each scheme has different resource demands and limits.

Thread per client:
Reserves one thread per connected client. The thread is allocated on connection establishment. This guarantees the permanent availability of execution resources for incoming requests. Only one concurrent request per client can be processed, do to the one thread per client limit.

Thread pool per client:
A pool of threads is assigned to each active connection. The number of possible active concurrent request is determined by the thread pool limits.

Thread per invocation:
Each incoming request gets assigned a new thread. Excessive amounts of connections or concurrent invocation request can exceed the available resources. This can be avoided by using thread pools with appropriate limits instead.

Thread per object:
Invocation requests to an object are only executed in a thread context dedicated to the object. Invocations are effectively serialized by this method.

4.2.1 Mutual exclusion and synchronization

Mutual exclusion
Some data structures and other resources are unshareable, usable only by one thread at a time. Access to those data structures is mutual exclusive. Primitives that can be used to enforce that are mutexes, reader-writer-locks and semaphores.

Synchronization
Threads have to pass information between them and may depend on the state of processing a different thread has reached. This requires two or more threads to be synchronized. Primitives for that are semaphores and conditional variables.

Conclusions

Thread manipulation heavily depends on the hardware structure and on the application demands. It has to be highly configurable at compile-time and at run-time. The ORB internal and external interface to thread operations should be uniform across different hardware and software platforms and should hide all system specific details from the ORB and application developer without limiting his ability to adopt thread handling to his needs. Basic uniform mechanisms to guarantee mutual exclusion and realize synchronization between threads are needed.

4.3 User-code challenges

Being support code, ORB's have to meet a third challenge – *user code*. The application programmer can not be expected to know about internal details of the ORB. It is therefore likely that the user code using an ORB is not optimized to the requirements of that particular ORB.

Two main problems can be identified:

Usercode live and dead locks
: Execution resources like memory and threads are occupied. Standard live and deadlock detection prevention and avoidance mechanisms can be used to handle those situations.

Resource usage
: Some invocation might consume extensive resource. This can be execution time, memory or other resources that are only available in limited quantities and have to be shared with other consumers.

Conclusions

The CORBA standard defines no support mechanism to address the describe problem areas. Any extension provided by the ORB will therefore be proprietary. Implementing mechanisms to provide features to address these problem areas should therefore be deferred until appropriate interfaces are part of the CORBA standard. Until then problems of this category have to be dealt with on a case-by-case basis by the developer in the application context.

5 Multi threading framework

Parallel execution requires the identification of code sequences that are independent and of the data structures that are shared. The shared data structures then have to be protected and the code sequences can be assigned to threads. Other requirements and interfaces to provide CORBA compliant support for multiple threads within the ORB have been identified in chapter 3 and 4.

5.1 Basic concepts

A invocation request goes through different stages as it progresses through the ORB. These stages can be viewed as parts of a pipeline (↗ Figure 5.1).

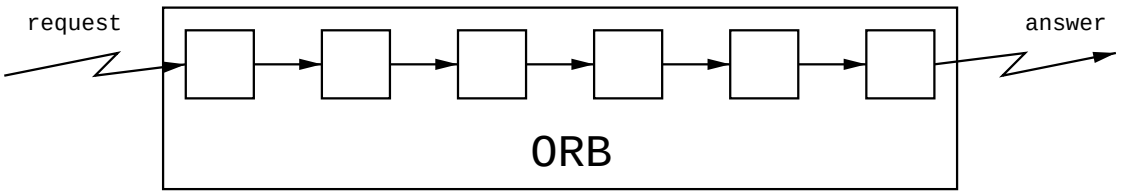


Figure 5.1: Execution pipeline

A request can not enter the next stage before processing it in the current stage has been finished. Each stage can process more that one request at a time parallel to and independent from other requests or stages. The implementation of a pipeline stage is not limited to software or bound to a specific execution place. It is possible to burn simple stages in hardware or have them executed on designated CPU's or nodes in a cluster.

Pipeline stages are represented by the *Operation* class (↗ Sect. 5.1.1). This class provides a uniform interface to start and execute the processing stage. For each request that is being processed in a stage a separate *Operation* object is being instantiated.

Control over how many requests a stage can process parallel and the execution resources for it are provided via instances of the *ThreadPool* class (↗ Sect. 5.1.3). Each *ThreadPool* object manages a specific pipeline stage (↗ Figure 5.2). It assigns threads to stages on request and manages the request delivery between stages.

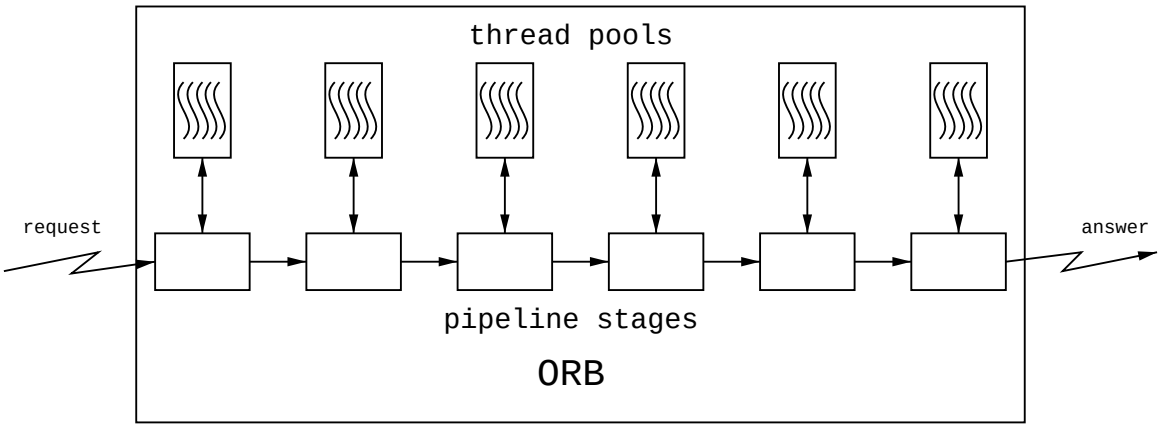


Figure 5.2: Thread pool controlled execution pipeline

A thread is represented by an object instance of the *WorkerThread* class (↗ Sect. 5.1.2). Assigning a *WorkerThread* object to a specific processing stage means assigning a *Operation* object to it for execution.

Requests are passed to the next stage by sending a message through a message channel (↗ Sect. 5.1.4). Messages contain references to the request, the destination stage, the message priority and possible additional scheduling information. Messages channels can reorder messages based on their priorities and their scheduling information (↗ Figure 5.3).

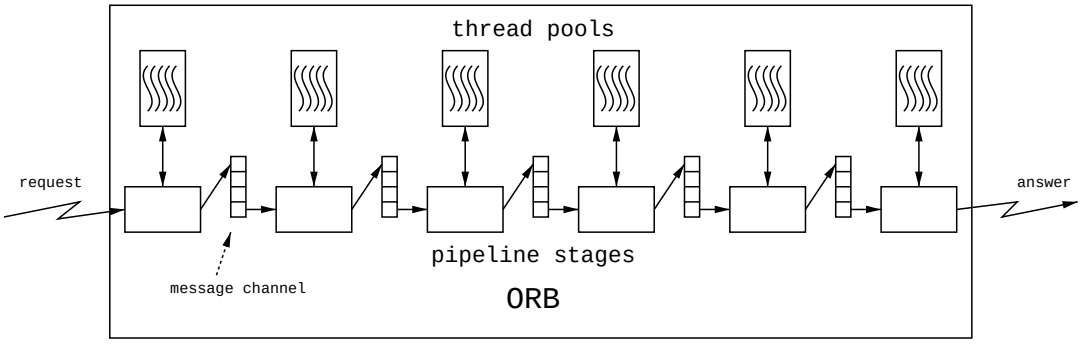


Figure 5.3: Full execution pipeline

A single stage of the execution pipeline consists therefore of the *ThreadPool* controlling the execution resources, the input message channel for receiving execution requests, a number of *WorkerThread* objects which can either be idle or busy in which case the associated thread executes a *Operation* object which processes a request. Figure 5.4 show a single pipeline stage with a message queue as input channel. The input channel does not have to be a queue. Any object that is derived from the *MsgChannel* class (↗ Sect. 5.1.4) and implements the necessary methods can be used as input channel.

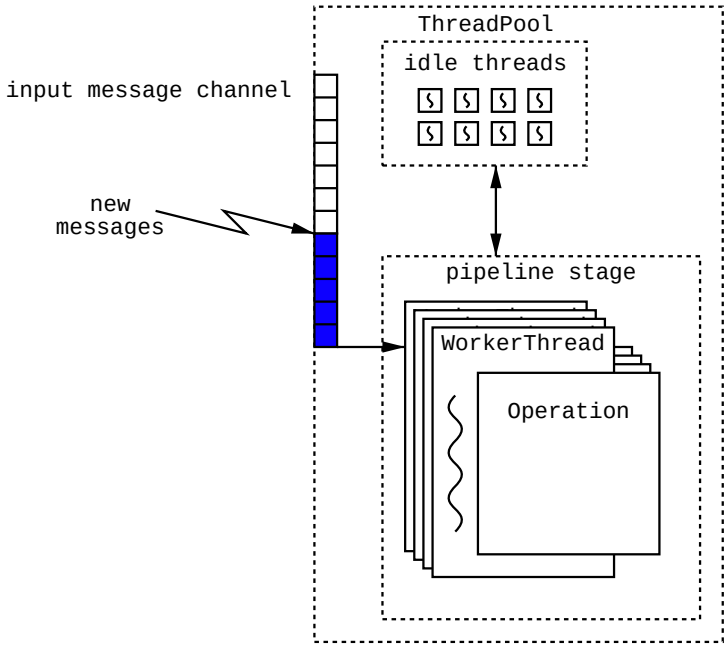


Figure 5.4: Single stage in the execution pipeline

5.1.1 Operations

An *Operation* object encapsulates code and state information associated with the request that is being processed in the current pipeline stage. An *Operation* object can therefore be in three different states:

`idle:`
: the operation has no data to process

`ready:`
: there is data to process but no thread has been assigned to execute the operation

`active:`
: a thread has been assigned to the operation to process the data

Operations delegate requests to other stages by sending messages to them and get new requests assigned by receiving them from a previous stage. They can therefore be classified into message producers and message consumers. The typical operation is both, it receives a message, processes the message and sends a message to the next operation.

Some operations have special roles and are either only consumer or producer. These special roles are mostly determined by the functions they provide.

- Input operations read data from an input channel and generate a new message. They are mostly producers.
- Output operations write data to an output channel. They are mostly consumers.

However, both operation types have to respond to control messages and may generate messages indicating error conditions.

UML representation

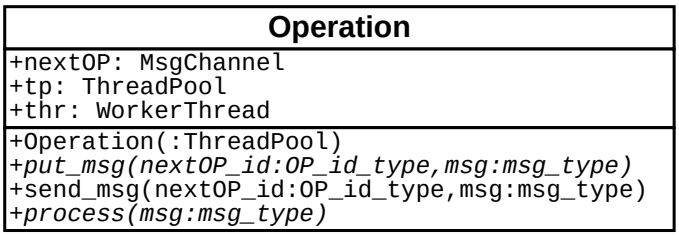


Figure 5.5: Operation class

Attributes

`nextOP (ro):`
: points to the operation that during normal processing will come next

`tp (ro):`
: the thread pool this *Operation* belong too

`thr (rw):`
: the thread that is currently executing this *Operation* or NULL

Methods

`put_msg:`
: Stores the message in an Operation private buffer for later evaluation

`send_msg:`
: Delivers a message to `nextOP`. Derived classes may overload this method to implement different message delivery strategies.

`process:`
: Evaluates the message and provides the implementation of the Operation. This is a pure virtual method and has to be overloaded with an actual implementation.

Several operations have special execution demands. Those demands may be application global or depend on a specific parameter set i.e. a file descriptor (FD) or a set of file descriptors.

global – at least one:
: More than one instance of this operation may exist. At least one of those instance has to be active, i.e. global error handler

global – exactly one:
: There is exactly one instance of this operation and it has to be active at any give time, i.e. global signal handler

parameter dependent – at least one:
At least one instance of this operation has to exist and be active for each value of the parameter, i.e. socket writer thread per FD/FD set

parameter dependent – exactly one:
Exactly one instance of this operation has to exist and be active for each value of the parameter, i.e. socket reader thread per FD/FD set

Messages can be delivered to an operation in two ways.

Passive operations will be activated by an external event. The message has to be passed into the *Operation* object and a thread has to be assigned to the *Operation* object to process the message. The *Operation* object has no control over when to execute. There is no knowledge over where the message originated or which message channel was used to deliver it to the current pipeline stage.

Active operations will actively request new messages from an input message channel. The message request method is allowed to block and wait until a new message arrives. The *Operation* object is in control of the input message channel. Especially, each *Operation* object in a pipeline stage can have its/their own input message channel, thereby allowing to address only a specific set of dedicated *Operation* objects and creating dedicated groups of *Operation* objects within a pipeline stage, i.e. the operations in a specialized stage for reading data from network via the socket interface can have dedicated input channels for each socket served.

5.1.2 Worker threads

The *WorkerThread* class provides a abstraction for the underlying *Thread* objects (↗ Sect. 5.3.2). It provides methods to assign *Operation* objects to it and to control their execution. A *WorkerThread* object always belongs to a *ThreadPool* (↗ Sect. 5.1.3). It is possible to move a *WorkerThread* to a new *ThreadPool* when it is idle or as the result of an synchronous message from one operation to the next operation.

A *WorkerThread* object has three states:

`transition:`
the internal state of the WorkerThread object is being manipulated. It is neither idle nor busy. Manipulation of thread internal information like assigning operations is only allowed while the thread object is in this state.

`busy:`
The thread is currently executing an Operation and is not available for other processing.

`idle:`
The thread is waiting for work and can be used to execute the next Operation

WorkerThread objects use the generic `process()` method (↗ Sect. 5.1.1) of *Operation* objects to execute them. They are pipeline stage independent. The main execution loop of *WorkerThread* objects is show in figure 5.6

WorkerThread objects can be told to exit from their main execution loop (their `_run()` method) and terminate the underlying *Thread* object by setting their state to terminate.

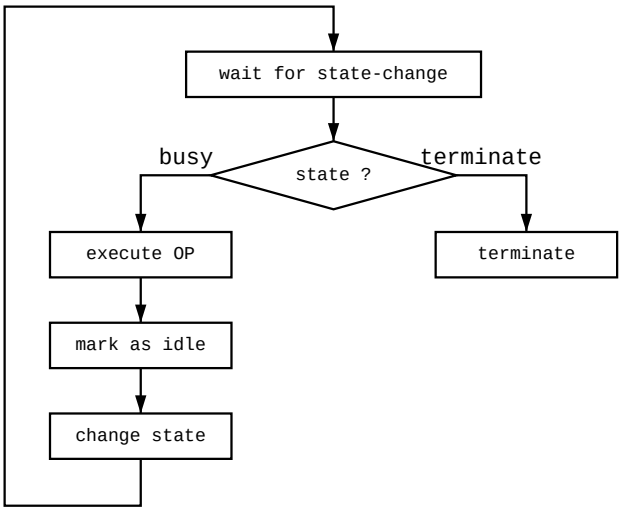


Figure 5.6: Thread main execution loop

UML representation

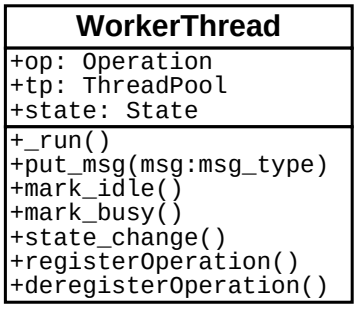


Figure 5.7: WorkerThread class

Attributes

`OP (ro):`
: points to the current operation object

`tp (ro):`
: the thread pool this *WorkerThread* belongs to

`state (rw):`
: active execution state

Methods

`_run:`
: Generic entry point into the main execution loop. Is only to be used by *ThreadPools* to start the thread.

`put_msg:`
: Delivers a message to the current operation.

`mark_idle:`
: Indicates that this thread is going into its idle state and is ready to perform work.

`mark_busy:`
: Indicates that this thread is switching to its busy state and will not be able for other processing. This method has to be called before assigning work to a thread. The thread will become unavailable for other operations.

`state_change:`
: Used for asynchronous notification of state changes. Has to be called in order for a state change to become effective. The thread will be waken if necessary and can perform whatever action was requested.

`registerOperation:`
: Attaches a instance of an *Operation* object to this thread. This is only informal and calls are only allowed in the idle and transition state. Processing will begin only after the appropriate state change has been set.

`deregisterOperation:`
: Detaches an *Operation* object instance from this thread. Only allowed in the idle and transition state.

5.1.3 Thread pools

ThreadPool objects encapsule the management of a group of threads. Each thread pool is responsible for exactly one operation type. It provides message routing, automatic, on demand thread creation, activation and termination.

Message routing

For passive operations, messages sent to a thread pool are automaticly routed to the next available operation or queued for later processing when necessary using an internal message queue.

For active operations, messages sent to a thread pool are delivered directly and immediately. No synchronization is performed. The operation has to implement its own message queue.

Thread creation, activation and termination

For passive operations the number of maximum idle threads, minimum idle threads and maximum threads are controlled and the running idle threads are adjusted when necessary based on runtime configurable parameters. Idle threads are assigned to operations on demand.

Active operations are constantly active and running. The associated thread is terminated when the operation returns from its `process()` method (↗ Sect. 5.1.1). Passive operations are terminated when they return from their `_run()` method which can happen only as a result of a state change to terminate (↗ Sect. 5.1.2).

UML representation

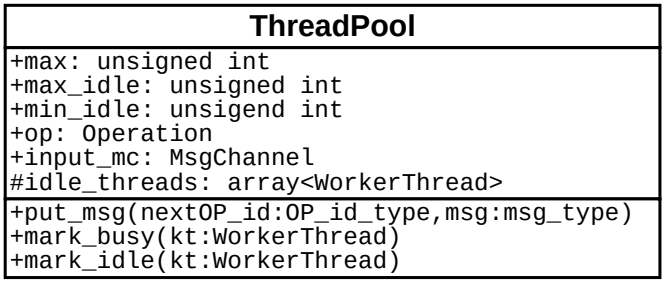


Figure 5.8: ThreadPool object

Attributes

`max (ro):`
maximum number of all running threads

`max_idle (ro):`
maximum number of threads in the idle state

`min_idle (ro):`
minimum number of threads in the idle state

`op (ro):`
Operation prototype

`input_mc (ro):`
Input message channels for passive operations.

`idle_threads (private):`
List of idle threads. Used internally to keep track of idle threads.

Methods

`put_msg:`
Deliver a message to the next free *WorkerThread* in this *ThreadPool*.

`mark_idle:`
Moves a thread into the pool of idle threads.

`make_busy:`
Sets a threads state to busy.

5.1.4 Message channels

A message channel is an abstract interface to deliver messages to objects (figure 5.9). Every object that can receive a message is therefore derived from the *MsgChannel* class (figure 5.10).

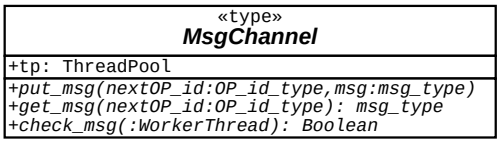


Figure 5.9: Abstract base class for message channels

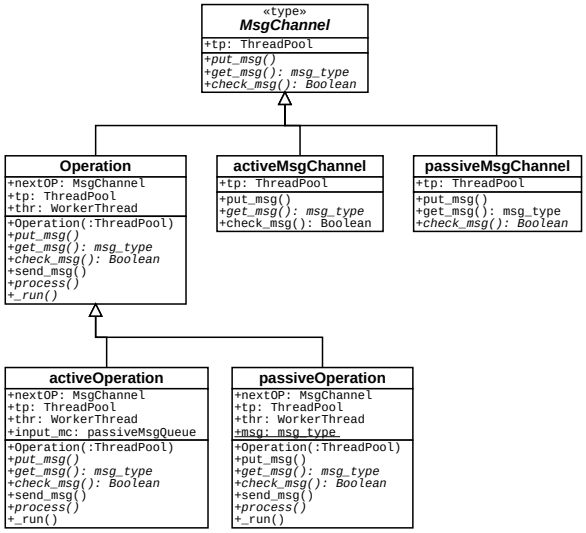


Figure 5.10: Message channel inheritance (parameters shortened)

5.1.5 Message queues

Message queues transfer and deliver messages from one operation to another. A message queue can deliver messages synchronously or asynchronously, it can reorder messages, change message priorities and determines how to route messages to the receiver. For the caller, message routing and delivery appears always to be transparent and asynchronous.

Synchronous message delivery

Takes effectively the form of a method call to a local *Operation* object. Since the new active *Operation* object belongs to a different thread pool, the old thread pool has to be notified of the change.

Asynchronous message delivery

The sender returns immediately without waiting for any result. The message will be processed in a different context. The typical control flow for this type is shown in figure 5.11.

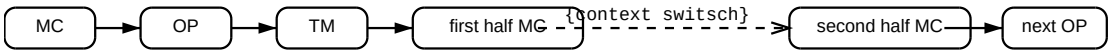


Figure 5.11: Control flow

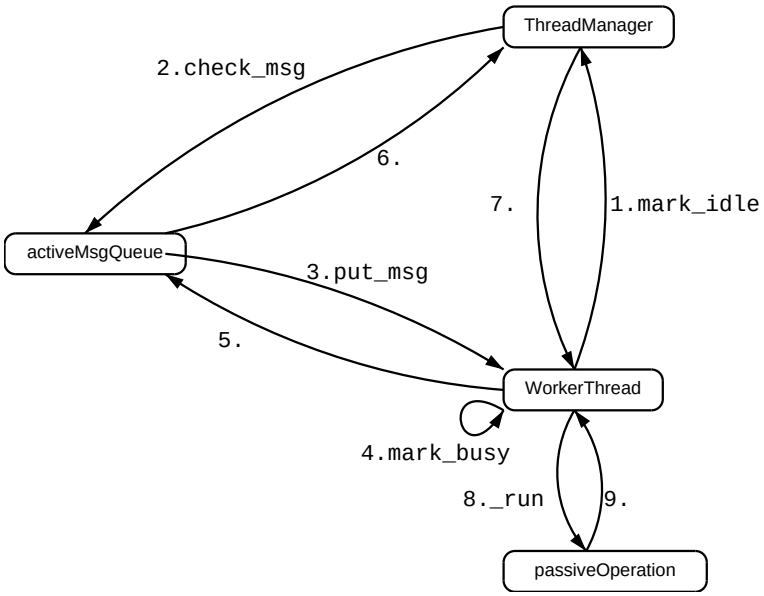


Figure 5.12: Interaction between active message queue and passive operation

Active message-queues and passive operation

A message queue that actively tries to deliver a message to an operation is realized in the *activeMsgQueue* object. Such a queue tries to request an idle thread from its thread pool, puts the message in the associated operation and marks the operation as runnable. The message is put into a wait queue when no idle thread is available. The wait queue is check each time a WorkerThread tries to enter its idle state and therefore becomes available for message processing (↗ Figure 5.12).

The message delivery delay is determined by the time it takes to acquire an idle thread. Idle threads can be acquired from a list of threads, by creating a new thread or by waiting for the next available thread. The parameters of the associated thread pool can be adjusted during idle thread acquisition.

Operations using the active delivery scheme work only on demand. Their processing algorithm is invoked via the `process()` method (↗ Sect. 5.1.1). Those operations are realized using the *passiveOperation* class as base class.

This message delivery scheme is applicable to thread pools where all operations share the same input message queue.

Passive message queues and active operation

A message send through a passive queue is always inserted into a wait queue. Operations actively try to get a message using the getMsg() method which blocks and waits when no messages are available (↗ Figure 5.13).

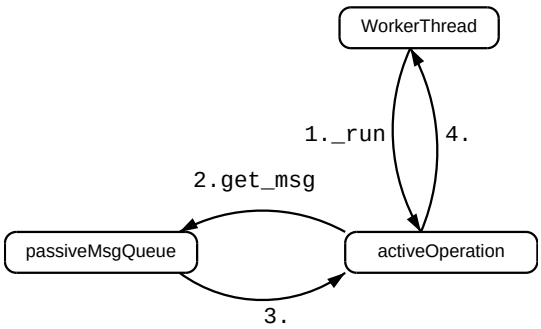


Figure 5.13: Interaction between passive message queue and active operation

The message delivery delay is determined by the time it takes to unblock a waiting operation. Adjustments to the parameters of the associated thread pool is not performed.

This message delivery scheme is applicable to single operations or groups of operations that have their own private input channel but share a common thread pool.

5.1.6 I/O operations

I/O operations constitute a special set of operations. They have to use system calls and library functions that have arbitrary limits and special semantics. Most operating systems provide a POSIX socket compliant API for standard network I/O other communication mechanisms and API's can be mapped to use socket like semantics.

Typical problems and limits of those API are:

- Some blocking system calls are not interrupted by socket destruction. This applies for `select()`, `poll()` and `accept()` on sockets that are in the listen state.
- Blocked system calls are woken up by signals even if the signal was handled by a different thread. This happens on most UNIX variants.
- the use of one thread per socket can exceed arbitrary limits on maximum allowed threads.
- Synchronous I/O multiplexing functions like `select()` and `poll()` limit the number of testable sockets and have large memory requirements
- Testing large numbers of sockets for pending events with one call takes considerable time
- Some operations systems provide special, non portable, non standard, high performance interfaces for asynchronous I/O

To provide a uniform interface for all types of I/O operations and to encapsulate the special knowledge associated with certain types of operations is I/O handling split into two components. The I/O functions are described by I/O channel objects that are derived from the *IOChannel* class and the I/O event handling and dispatching is handled by I/O dispatcher operations.

I/O Dispatcher

An I/O Dispatcher is a special purpose *Operation* that manages the event checking for a group of I/O channels. I/O channels can be standard sockets or also any other communication mechanisms that can be mapped onto the required semantics. Events are handled by I/O handlers. Channels have to be registered with the I/O dispatcher, providing the channel identifier, the type of event and the handler to be notified by a callback in the case of an event. Possible events are:

Accept:
the socket received a connection establishment request

Read:
data is available to be read

Write:
next write operation on this channel will not block

HungUp:
the channel has been disconnected

Error:
an error occurred on this channel

Event callbacks always indicate the channel and the event, allowing to use one event handler for a set of event types and channels. Only channels that use the same event checking mechanism are allowed to be grouped together.

I/O Dispatcher objects being *Operation* objects can be handled by a special version of a *ThreadPool*. This allows to distribute the load of event checking across a fixed number of threads which can be adjusted as load increases or decreases. It allows to group channels according to their priorities and assign them to threads of special priority groups within the *ThreadPool*. Furthermore, special I/O dispatchers and *ThreadPool* for accept, read and write events can be used to impose different limits and to realize different event reaction profiles.

5.2 Basic patterns

A in depth analyses of the chosen reference platform has lead to the identification of a number off patterns that are relevant for parallel executed code. All those patterns are typical for and widespread used in CORBA ORB's. The following sections explain their typical usage and list requirements for the framework to support their use.

5.2.1 Reference counting

Reference counting is used throughout the implementation to ease automatic resource tracking for objects and to implement automatic references according to CORBA standard ([7] Sect. 4.3.2)

Reference counting is a variation of smart pointers. For the scope of this paper a reference is considered to be a pointer to a data structure containing a reference counter. This data structure is from now on being referred to as an object. The reference counter keeps track of how many references exist to this object. The object can not be destroyed before the reference counter reaches zero.

There are two operations to manage a reference:

duplicate() increments the reference counter and returns a new reference to the caller
release() release a reference and destroys the associated data structure when all references have be released, that means when the reference counter has reached zero

The following code sequence illustrates the basic mechanism used in MICO:

```
static ServerlessObject* _duplicate (ServerlessObject *o)
{
   if (o)
       o->_ref();
   return o;
}

void release (ServerlessObject *o)
{
    if (o && o->_deref())
    {
        delete o;
        o = NULL;
    }
}

void
CORBA::ServerlessObject::_ref ()
{
    _check ();
    ++refs;
}

CORBA::Boolean
CORBA::ServerlessObject::_deref ()
{
```

```
    return _check_nothrow() && --refs <= 0;
}
```

The _check and _check_nothrow methods guarantee the validity of the referenced objects.

Reference counter integrity

The reference counter variable can be accessed simultaneously by different threads. It is therefore necessary that the increment and decrement-and-test operations are atomic to protect the integrity of the counter.

Reference integrity

The above code sequence has a potential race condition where the object pointer could be corrupted between the first test and its subsequent use. Consider the execution sequence in figure 5.14.

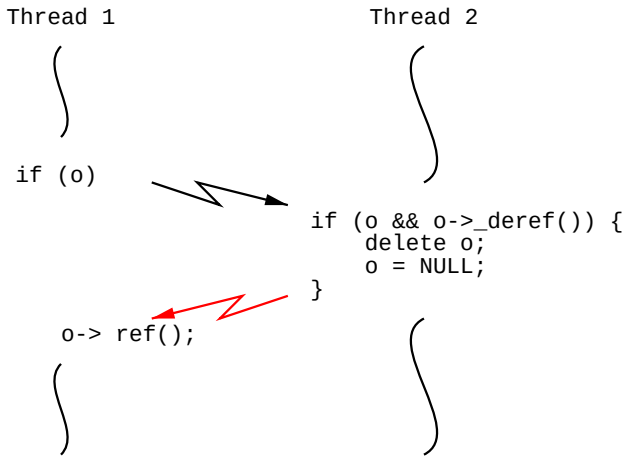


Figure 5.14: Reference corruption

This situation can only exist when the object reference that is being duplicated has been obtained without incrementing the reference counter.

5.2.2 Object life cycle

Reference counting guarantees the existence of objects as long as there are references to them. That leaves two problems:

1. There might be states where the primary usefulness of an object has ended and no operations should be allowed any longer on it, but where it can not be deleted because there are still pending references on it.

2. A sequence of instructions assumes that the state of an object if constant.

A good example for that is an ORB in a multi thread environment:

1. initialization – the ORB is created and initialized, no invocations are permitted during that period
2. operational – the ORB is fully functional, invocations are being processed
3. init shutdown – the ORB shutdown has been requested and is pending
4. shutdown – the ORB has been terminated, no invocations are permitted
5. destroy – the ORB object has been deleted

In this example, invocations might assume that the ORB is not changing its state while executing special functions i.e. marshaling or demarshaling of parameters. Also invocations might be active at the time of shutdown which can not be canceled. These invocations will have references to the ORB and might try to access the ORB upon return. It is therefore not possible to simply remove the ORB object after shutdown.

The required behavior can be realized as an extension to the normal reference counting with the introduction of an object state and an extra counter for references in the active state. Such a reference is called an *active reference* from now on.

The object states follow closely the example above with the states being: init, active, init_shutdown, shutdown (5.15).

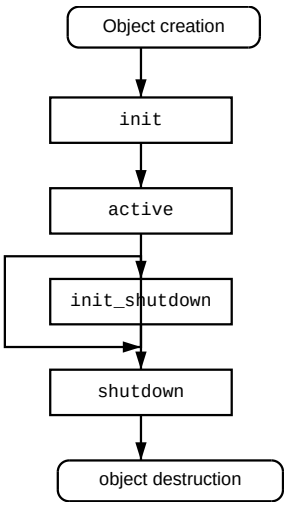


Figure 5.15: Object life cycle

A caller can acquire an active reference only when the object is in its *active* state and the object can not enter the *shutdown* state before not all active references have been released.

The states have the following meaning:

`init:`
: Only one thread can be active in the *init* state. Transition from *init* to *active* can occur at all times. Threads wishing to acquire an active reference have to wait until the object enters it *active* state.

`active:`
: Any number of threads can access the object in this state. A thread depending upon the object staying in this state has to acquire an active reference. The attempt to acquire an active reference will only succeed if the object is in this state.

`init_shutdown:`
: This state indicates that the object is about to be shut down. No action is being taken to actually change the objects status. After the object state has been set to *init_shutdown* or *shutdown* active references will no longer be granted.

`shutdown:`
: The object is in the process or has been shut down. Only one thread is allowed to access the object in this state. The attempt to change the object state to *shutdown* will only return after all active references have been released and only the first caller will indicate success upon return.

`destroy:`
: All references to this object have been released. The object memory has been freed. Any attempt to access the object will result in undefined behavior.

5.2.3 Thread unsafe STL primitives

CORBA mandates the use of standard template library (STL) containers for many data types and structures in a C++ CORBA implementation. The STL guarantees the thread safetiness of its containers only for reading access. Modifying the content of an STL container automaticly invalidates all iterators that uses this container. That leads to the conclusion that a multiple reader/single writer lock is required to serialize access to shared STL container. A typical example for such a structure is the active object map in the POA.

Some data structures implemented using STL containers are mostly used in a fashion where a write operation might be necessary directly after a read operation and no modification in between is allowed. Code sequences like that need full and exclusive access to container for both operations without interruption. It is therefore not possible to use a reader/writer lock, release the reader lock and reacquired the writer lock. The use of a mutex or semaphore is required.

5.2.4 Thread synchronization

Sometimes one thread has to wait until an other thread has generated results e.g. waiting for the results of an invocation. Synchronization can be archived using s semaphores or conditional variables. A semaphore counts the number of waiting threads (wait operations) and the number of pending notifications (post operations). No notification will be lost. A conditional variable guarantees the integrity of state variable used and the reliable notification of the waiter of its change.

5.2.5 Recursive locks

Interactions between ORB and user code as well as recursive calls during reference resolving lead to recursive calls of methods that contain critical sections. The locks protecting those sections need to be recursive to avoid deadlocks.

5.2.6 Scoped Locking

"Scoped Locking" is a technique that can be used to ensures that a lock is acquired when control enters a scope[1] and the lock is released automatically when control leaves the scope. It guarantees robust locking, i.e. locks are always acquired and released properly when control enters or leaves critical sections. It requires that the critical section represents a method or block scope. Additional scopes can be introduced to meet that requirement, but they can obfuscate code and thereby can make the source code less readable and understandable.

It is usually not a good design to have critical sections with multiple return paths, but when adapting third party or old code for multi threading this situation frequently occurs. If locks are acquired and released explicitly, it is hard to ensure all paths through the code release locks. Code maintenance might also frequently change code and return paths, which can lead to hard to find problems caused by missing locks or dead locks.

Constructs like:

```
boolean foo() {
    autoLock l(m);

    /* modify i */

    return (i != 0);
}
```

are problematic, because it is not defined whether the lock is released before or after the return expression has been evaluated.

Only the reader-writer-lock and the mutex are target for scoped locking. Semaphores and conditional variables are usually used in more complex scenarios where scoped locking is not applicable.

[9] favors the use of parameterized types or polymorphic lock objects over separate different auto-locks for each lock type. Reasons given there are:

> Defining a different Guard class for each type of lock is tedious, error-prone, and may increase the memory footprint of the application or component.

The author has found, while testing different approaches, that while it indeed can be tedious to change lock types and all related auto-locks, strict type checking provided by the compiler helps locating lock usage mistakes and code optimization performed by the compiler produces better results than for the other two variants.

1 scope refers to programming language scope like the block-begin { and block-end } construct in C++

5.2.7 Double-Checked Locking

Often execution of critical sections depends one various conditions.

Consider the following example:

```
void foo::bar(int &cond)
{
  mutex.lock();
  if (cond == 0) {
    /* critical section begin */
    ...
    cond = 1;
    ...
    /* critical section end */
  }
  mutex.unlock();
}
```

"Double-Checked Locking" is useful under the following conditions:

- the condition needs protection against modification of tested elements
- the critical section solely depends on the result of the condition
- modification of elements of the condition is protected by the same lock as the current block

Speed gain can be achieved when execution of the critical section is rare. The improvement stems from reduced overhead for locking operations and from avoiding unnecessary waiting for lock availability.

When using "Double-Checked Locking" the condition is checked without acquiring the mutex first. This could lead to a situation where two ore more threads try to enter the critical section at the same time while only one is supposed to do so. In a next step the mutex is therefore to be acquired and the condition has to be reevaluated since it status could have been changed.

C++ example for modified code segment:

```
void foo::bar(int &cond)
{
  if (cond == 0) {
    mutex.lock();
    if (cond == 0) {
      /* critical section begin */
      ...
      cond = 1;
      ...
      /* critical section end */
    }
```

```
    mutex.unlock();
  }
}
```

Double-Checked Locking is problematic and has to be evaluated on a case-by-case basis if:

the condition is complex
: The time spend for evaluating the condition twice might be worse than original overhead and long conditional statements are automaticly non atomic.

modification of elements of the condition is non atomic
: Non atomic modification can create interim states with invalid states leading to various race conditions.

the condition leads to frequent execution of the critical section
: Overhead for checking the condition twice negates the desired speed improvements.

5.3 Interfaces to thread primitives

Almost each thread package has its own interface for thread initialization and control and a different set of and a special interface to its synchronization primitives.

In order to be able to support a wide variety of operations systems and thread packages it is necessary to define a common interface.

The most commonly found interfaces are the POSIX Threads Extensions (1003.1c-1995) and the X/Open interfaces as defined in the single UNIX specification.

The interface definition for the synchronization primitives and the thread abstractions used here are derived directly from those standards.

5.3.1 Synchronization primitives

Synchronization primitives are used to coordinate access to share resources and to notify waiting threads of status changes. The primitives present here are not only for the ORB internal use, the application is free to employ them as well. The rtCORBA specification defines a mutex. Using a mutex or a semaphore each off the other primitives described here can be constructed.

The presented synchronization primitives are not always the best solution. They have been employed extensively in the prototype implementation knowing well that other better suited replacements like optimistic locking or non-blocking synchronization exist. The reasons for choosing these have been explained in Sect. 5.2.3 till 5.2.5.

Mutex

A *Mutex* object can have two states, locked and unlocked. It is used to regulate the access to a shared resource. A thread that wishes to the resource first has to lock the mutex and unlock

it when the resource is no longer used. No other thread is allow to acquire (lock) the mutex if it has already been locked by a different thread. A *Mutex* object can be initialized in the locked or unlocked state. Two types of mutexes are supported, recursive and non-recursive.

Available operations:

`lock()`:
: This operation locks the mutex object. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object in the locked state with the calling thread as its owner.

  If the mutex is non-recursive, self deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

  If the mutex is recursive, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not been locked or a mutex which is unlocked, an error will be returned.

`trylock()`:
: This operation is identical to lock() except that if the mutex object is currently locked (by any thread, including the current thread), the call returns immediately.

`unlock()`:
: Releases the mutex. In the case of a recursive mutex, the mutex becomes available to other threads when the count reaches zero.

C++–Mapping

```
class Mutex {
public:
    typedef enum { Normal, Recursive } Attribute;
    typedef enum { NoError, AlreadyLocked, TryAgain, Invalid, Fault,
                   DeadLock, UnknownError } ErrorType;

    Mutex( Boolean locked = FALSE, Attribute attr = Normal );
    ~Mutex();

    ErrorType trylock();
    void lock();
    void unlock();
};
```

Semaphore

A *semaphore* can be used much like the mutex to coordinate access to a shared resource. It is here implemented as a counting semaphore. If used in a mutex like fashion a value of one

means that the resource is available and can be locked by performing the `wait` operation. A semaphore can be used for cross thread synchronization where `post` indicates that a certain event has occurred and how often it happened and `wait` waits until the event occurs. Since it is implemented as a counting semaphore, no event will be lost.

Available operations:

`wait():`
: This operation locks the semaphore object by performing a semaphore lock operation on it. If the semaphore value is currently zero, then the calling thread will not return from the call to wait() until it locks the semaphore.

`trywait():`
: Locks the semaphore object only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

`post():`
: Unlocks the semaphore object by performing a semaphore unlock operation on that semaphore. If there are threads waiting for this semaphore to become available, then exactly one thread is being unblocked.

C++–Mapping

```
class Semaphore {
public:
    typedef enum { NoError, NoPermission, TryAgain, SemInvalid,
                   Interrupted, UnknownError } ErrorType;

    Semaphore( unsigned int val = 0);
    ~Semaphore();

    void wait();
    Semaphore::ErrorType trywait();
    void post();
};
```

Conditional Variable

A conditional variable indicates that a certain event has occurred. The *CondVar* object is used to serialize access to a conditional variable and reliable notify other thread about state change of a conditional variable. Operations performed on the *CondVar* object do not change the state of the conditional variable.

Available operations:

`wait(Mutex):`
: This operation is used to block on a condition variable. Is has to be called with mutex locked by the calling thread or undefined behavior will result.

This operation atomically release mutex and cause the calling thread to block on the condition variable; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to signal() or broadcast() in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

`timedwait(Mutex, timeout):`
: This operation is the same as wait() except that an error is returned if the time specified by timeout passes before the condition is signaled or broadcasted.

`signal()`
: Unblocks at least one of the threads that are blocked on the condition variable (if any threads are blocked).

`broadcast():`
: unblocks all threads currently blocked on the condition variable.

C++–Mapping

```
class CondVar {
public:
    CondVar();
    ~CondVar();

    Boolean wait( Mutex &_mutex );
    Boolean timedwait( Mutex &_mutex, unsigned long tmout );
    void broadcast();
    void signal();
};
```

Reader/Writer Lock

A reader/writer lock coordinates access to a shared variable that can be read by multiple threads, but can be written only by a single thread.

Available operations:

`rdlock():`
: Applies a read lock to the read-write lock object. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the rdlock() call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on the object at the time the call is made.

`tryrdlock()`:
: Applies a read lock as in the rdlock() operation with the exception that the operation fails if any thread holds a write lock on the object or there are writers blocked on the object.

`wrlock()`:
: Applies a write lock to the read-write lock object. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock. Otherwise, the thread blocks (that is, does not return from the wrlock() call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

`trywrlock()`:
: Applies a write lock like the wrlock() operation, with the exception that the operation fails if any thread currently holds rwlock (for reading or writing).

`unlock()`:
: Releases a lock held on the read-write lock object. Results are undefined if the read-write lock is not held by the calling thread.

C++–Mapping

```
class rwLock {
public:
    rwLock();
    ~rwLock();

    void rdlock();
    void wrlock();
    void unlock();
};
```

Primitives not offered by a specific package can usually be emulated using others.

5.3.2 Thread abstraction

The thread abstraction object contains all thread specific attributes. The existens of an *Thread* object does not automaticly imply that a thread running. It is merely a wrapper to create and destroy threads through an object context and to store all thread attributes in a centralized place.

UML representation

Attributes

`Id (ro)`:
: System thread identification, is used to identify the thread to the used thread package and operating system

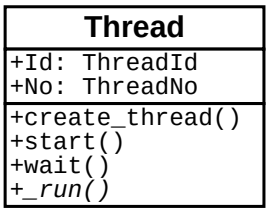


Figure 5.16: Thread class

`No (rw):`
A thread number assigned by the application, is used to identify threads to other objects.

Methods

`create_thread:`
Creates a new thread in a blocked state.

`start:`
Unblocks a newly create thread so that it can run.

`wait:`
Waits until the associated thread has terminated.

`:run:`
Main entry point for the actual code executed by the thread.

Thread cancelation

Synchronous Thread cancelation allows a thread to terminate the execution of any other thread in the process at any point in it execution. Terminating a thread in this manner while it has resources allocated will prevent the deallocation of those resources. It is therefore not being used.

Asynchronous Thread cancelation allows a thread to be terminated at special predetermined points. It has to made sure that no resources are hold by a thread when it reaches those cancelation points. Asynchronous cancelation capabilities are integrated in the main execution loop of the *WorkerThread* object (↗ Sect. 5.1.2).

5.4 Atomic operations

Not all operations on data need to be fully synchronized. Simple atomic operations like integer increment or decrement–and–test already exists on most CPU architectures in an uni-processor configuration and are many times faster than using system provided synchronization around non atomic version. Most architectures also provide fast and simple mechanisms to implement those operations in multi-processor configurations. Most compilers and high level languages provide mechanisms to embed target system specific assembler instructions that can be used to achieve that goal without the overhead of full blown synchronization primitives. Those assembler constructs are platform dependent but operation system independent. It is also

desirable to have generic versions of those operations, which can be used on platforms where the proper assembler sequences are not yet known or where the desired operation can not be performed atomically without complex synchronization.

The following integer operation need to be available:

- decrement
- increment
- decrement-and-test

All operations can be implemented atomicly by surrounding the non-atomic version of the operation with a mutex. This is the worst and slowest possible solution and should only used when all other methods are impossible. C++ implementation example:

```
class atomic_integer {
private:
  Mutex m;
  int   i;

public:
  Boolean decrement_and_test()
  {
    Boolean res;

    m.lock();
    i--;
    res = (i != 0);
    m.unlock();
    return res;
  };
}
```

The above solution not only uses slow and complex synchronization primitives, those primitives can also block. A much better and non-blocking solution would be:

```
class atomic_integer {
private:
  int   i;

public:
  Boolean decrement_and_test()
  {
    int aux;
    do {
      aux = i;
    } while (!CAS(aux, i, aux-1));
    return (aux-1) != 0;
```

```
  }
}
```

The CAS operation implements an atomic compare-and-store, which sets the second argument to the value of the third argument when the first and second argument are equal. Some CPU's provide a atomic CAS or equivalent instruction direct (e.g. mc68k-series, i8x86 platform CMPXCHG-instruction), on other platforms can it mostly be implemented using a short sequence of simple instructions.

6 Prototype implementation

The previous chapter introduces a generic framework for multi threading a CORBA implementation providing thread pools and using message passing as intra thread communication mechanism. The framework is designed to achieve high scalability with low management and communication overhead. To prove that the described framework can deliver on its promises a prototype has been implemented.

6.1 Overview of MICO

The MICO ORB [10] was used as platform for the reference implementation. MICO is an OpenSource [11] CORBA implementation from the University of Frankfurt, originally designed and implemented by Arno Puder and Kay Römer. Version 2.2.7 was recently certified for CORBA 2.1 compliance by the Open Group [12].

One design goal of MICO was to be easily extensible, as detailed in Kay Römer's thesis [13]. With its micro-kernel approach, the MICO core provides abstractions for object adapters, transports and schedulers. By derivation from abstract base classes and using operating system provided dynamic loading capabilities, new components can be plugged in at runtime.

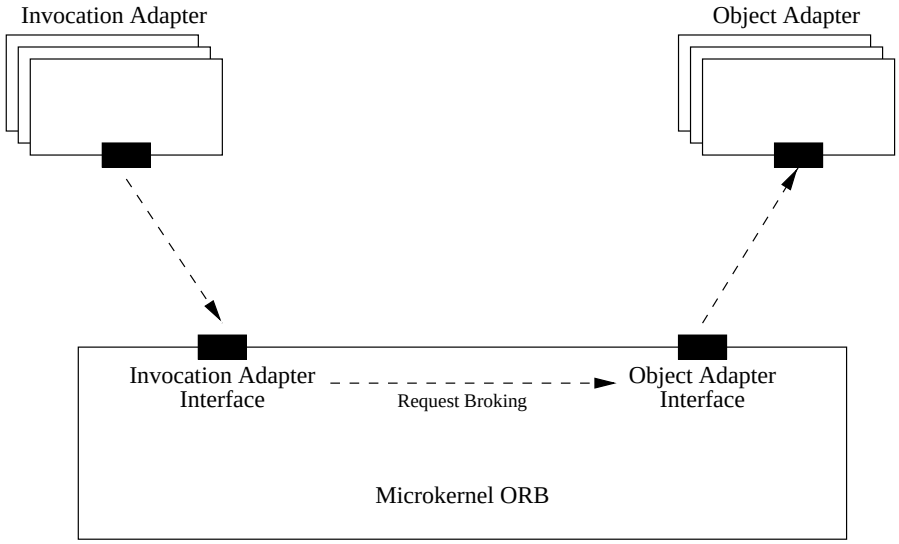


Figure 6.1: The MICO mikrokernel ORB

The micro-kernel ORB itself is reduced to the "broking" of requests. After a request is passed

to the ORB from an invocation adapter (like the Dynamic Invocation Interface (DII) or Static Invocation Interface (SII)), it simply tries all object adapters and asks whether they feel responsible for the request (↗ Figure 6.1).

An object adapter is not restricted to adapting local objects: sending a request to a remote server satisfies the object adapter concept – the location of an appropriate servant – just as well. Consequently, MICO uses IIOP proxy object adapters to address remote objects on the client side, and an IIOP server invocation adapter on the server side (figure 6.2). With this design, the ORB is not aware of an object's remoteness – a remote invocation is merely a side effect of passing the request to the right object adapter.

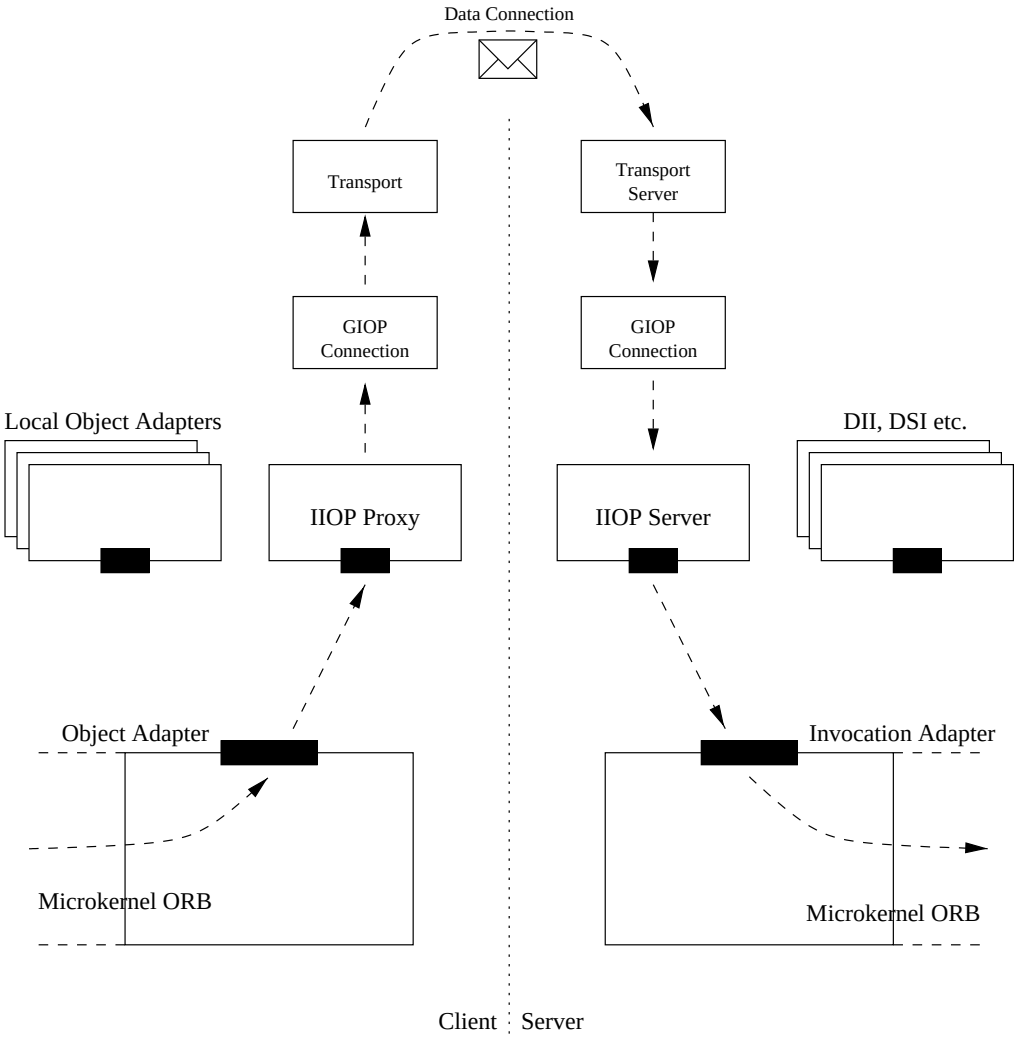


Figure 6.2: Remote invocation with a micro-kernel ORB

While IIOP proxy and IIOP server implement IIOP, they do not transfer data by themselves, but use General Inter-ORB Protocol (GIOP) connection objects to perform GIOP encoding,

and finally use instances of *Transport* and *TransportServer* for unidirectional[1] data transfer.

With this modular approach, the ORB is indeed easily extensible, as each component can be replaced. A new transport, for example one that compresses GIOP data before sending, can be added just by subclassing the Transport class and providing methods for reading and writing data.

6.2 Interface definitions

The interface semantics as described in this chapter are in most case clarifications and extensions for multi threaded use of the existing interfaces as defined in [13].

6.2.1 ORB interfaces

Invocation request and parameters are passed to and from the ORB in the form of *Thunk–objects*[2]. Such an object encapsulates the invocation request, its parameters and all information necessary to evaluate the request. The thunk is filled with values as it progresses through the object-adapters and the ORB.

Invocation-adapter interface

MICO uses the so called invocation-adapter interface to initiate remote method calls at the ORB interface. This interface is used indirectly by the Dynamic Invocation Interface (DII) and the Static Invocation Interface (SII). Server side transport modules also use this interface to deliver received request to the local ORB. The invocation-adapter interface theoretically is already asynchron (↗ [13]). MICOs object-adapters and transport modules using this interface however make a lot of assumptions about execution order and often violate or ignore the asynchronous nature of this interface.

The following methods are supported:

`ORB::invoke_async(`O`, `T`)`
: Initiates a method call at the ORB interface using as parameters the target object reference O and the method parameter thunk T and returns a ORB invocation request handle H.

`ORB::cancel(`H`)`
: Cancels the request indicated by a invocation request handle H.

`ORB::wait(`H`)`
: Waits for the result of an invocation indicated by a invocation request handle H and blocks until it has either arrived or the request has been cancel.

`ORB::get_invoke_reply(`H`)`
: Fetches the result of an invocation request H from the ORB.

1 CORBA 2.3 introduces bidirectional GIOP, so *Transport* and *TransportServer* will ultimately have to provide bidirectional data transfer, too.

2 The term *Thunk* for use with MICO internals is introduced in [13].

The challenge for this interface is that `cancel`, `wait` and the actual requested method might be executed at the same time and that `cancel` and `wait` have only limited knowledge of the current state of the invocation. `cancel` has to notify waiting threads and send a cancellation request to the server. It cannot delete internal state information for the request because a different thread might still need access to those information. The request handle has therefore to remain valid until `get_invoke_reply` has been called which will mark the completion of the request on the client side. Incoming results for already canceled requests will be discarded.

Object-adapter interface

The object-adapter interface is used by the ORB to delegate the execution of the invocation request to the responsible ORB component. That can either be a concrete object-adapter or a transport module.

This interface is traversed for every invocation at least twice. It has therefore to be efficient and multi thread safe to avoid unnecessary synchronization between threads.

The following methods are provided by object-adapters and transport modules:

`has_object(`O`)`
: This method is used by the ORB to query an object-adapter or transport module whether it is responsible for a specified object reference O or not. In the worst case this requires to check all registered object-adapters. No operations that might modify the object-adapters list are allowed during the entire query process.

`invoke(`H`, `O`, `T
: Hands the invocation request to the object-adapter. The return of `invoke` gives no indication about the current execution state of the invocation request. The callee can make no assumption of the state of the object reference O or the request thunk reference T. Both references can be assumed to be valid after `invoke` returns only if the callee claimed ownership (↗ Sect. 5.2.1) before returning.

`cancel(`H`)`
: Indicates that the specified invocation H should be canceled. The internal status information for the request will be marked accordingly, but no information associated with the invocation will be delete until the invocation itself indicates its return. Any return parameters will be ignored and discarded.

In standard MICO, requests passed from the ORB to an object-adapter are identified by their message id. To avoid extensive overhead for mapping message request id to ORB internal data structures a single cache structure had been used in MICO. This structure had to been removed for the multi threaded implementation since it was ineffective when more that one outstanding requests were active. To avoid the permanent mapping of message ids to internal references, the interface has been changed to use ORB request handles instead of message ids. ORB request handles are for ORB internal use only and carry no meaning outside of the ORB scope. Mapping to ORB internal references can be performed in $O(1)$ time. The following ORB methods provide mappings for ORB request handles to and from normal CORBA request message id's:

`ORB::get_msgid(`H`)`
: maps the ORB request handle H to its message id

`ORB::get_orbid(`I`)`
: maps the message id I to its ORB request handle

Invocation results are returned to the ORB asynchronous through the ORB's `answer_invoke` method.

Invocation processing and the invocation table

Because of the asynchronous nature of the object-adapter interface has the ORB to keep track of the active invocation requests that have return parameters. Invocation requests without return parameters are not tracked. For that purpose a list of currently active invocations is maintained with entries in the form (I, T)[3] consisting of the invocation message id I and the invocation request thunk T.

From the ORB's perspective a method call is executed in three basic steps. In the first step accepts the ORB the request from an invocation generating component and delegates it to the responsible component:

1. accept a method call $(O, T_{in/out})$ with `invoke_async`
2. generate a new message id I, a new handle H and build new ORB request thunk T from $(O, T_{in/out})$ and insert (I, T) into the invocation table
3. find the responsible component by calling `has_object` on all registered object-adapters and delegate the invocation by calling its `invoke` method.
4. return H as result

The object adapter has now either started to execution of the requests or send an invocation request to an other server. The request is now being processed and results are returned to the ORB:

5. the ORB accept return results (T_{out}) for the request H with `answer_invoke`
6. copy the relevant information from (T_{out}) to the $T_{in/out}$ associated with H
7. notify the invocation generating component of the invocation completion

In the third and final step, the invocation generating component obtains the results with `get_invoke_reply(`H`)` for the invocation request H:

8. wait until the invocation has been completed or canceled
9. remove (I, T) from the invocation table

3 The description given in [13] is not correct for current MICO versions

10. return invocation status

All steps can be executed in part in parallel. Protecting the involved data structures namely the invocation table and the invocation request thunk T appropriately is therefore of the outmost importance.

Access to the invocation table is managed by three methods:

`add_invoke(`T`)`
: insert a new request into the invocation table

`get_invoke(`I`)`
: return the parameter thunk for invocation I

`del_invoke(`I`)`
: remove the parameter thunk for invocation I

Conversion of a ORB request handle H into its parameter thunk T does not require access to the invocation table.

Request cancellation can happen at any time but not before step 2 has been completed. T will be marked as canceled, the invocation generating component will be notified and step 6 will discard any return results.

6.2.2 Scheduler interface

The scheduler is the central control instance in the MICO ORB. It has been designed with the intention to be usable in multi and single threaded environments. However, some of the assumptions made concerning its basic working principles are either not applicable or in other instances do not scale well in multi threaded environments.

Closely translated from [13], MICO's scheduler is designed to have the following characteristics:

- works in single and multi threaded environments
- interface is independent from implementation
- allows implementation of multiple scheduling algorithm

According to [13], the scheduler also makes the following assumptions about subtasks:

1. Code of parallel executable subtasks is dividable in two categories:
   - waiting operations, which wait for a specific event
   - processing operations, those are all operations that are no waiting operations
2. Time required for executing each sequence of processing operation of a subtask is negligible small
3. Only a limited number of events exists

   To assumption 1: Some operations can not immediately be identified as processing or waiting operations, but are a mixture of both. Input/Output operations for instance can under some circumstances block and wait until the input or output operations has been completed. Such mixed operations however can be transfered into a sequence of pure waiting and processing operations.

This assumption holds in single and multi threaded environments.

> To assumption 2: "negligible small" means that for any arbitrary but fixed time span $\Delta t > 0$ the time for executing any sequence of processing operations of all subtasks is less or equal Δt. A sequence of processing operations not satisfying this assumption can be transfered into a sequence of processing and dummy wait operations that meets all assumptions.

In multi threaded and especially multi processor environments, processing operations can be executed at exactly the same time. Therefore interactions and overlapping access to shared data can not be ignored however small the time frame might be. Execution time has to be considered for parallel executed operations but not for sequential executed operations.

> To assumption 3: Basicly all events could be mapped to a synchronization primitive like a semaphore. Eventually, some events will happen so often that is makes sense to provide them directly.

This assumption holds in single and multi threaded environments.

However [13] concludes from this that:

> A CORBA-system consists of different subtask that have to be processed parallel. With assumption 1 each program sequence of those subtask can be divided into blocks, beginning with a wait operation followed by processing operations. Since each subtask is executed sequential, only one block of each subtask is active (i.e. being executed). With assumption 2 the time for executing the processing operation is negligible, therefore at any given time one waiting operations per subtask is active and ready to execute the associated block on event entry.

With the limitation of assumption 2 the above is only valid for dependent, sequential executed blocks, but not for independent and *possibly* concurrently executed blocks.

Furthermore, it is desirable to execute more than one instance of a particular subtask at any given time to achieve better scalability on multi processor systems and to exploit priority dependent scheduling in preemptive environments. This leads to more than one waiting operation per subtask.

That leads to the following modified assumptions for the scheduling operations:

1. Code of parallel executable subtasks is dividable in three categories:
   - external waiting operations, which wait for a specific external event being signalized to the application
   - internal waiting operations, which wait for a specific internal event being signalized to a specific thread or group of threads
   - processing operations, those are all operations that are no waiting operations
2. Time required for executing each sequence of processing operation of a subtask is usually small, but not negligible

3. Selection of the next executable thread is performed outside the scope of the application.
4. Only a limited number of external events and event originator types exists
5. Only one internal event exists

To assumtion 4: External events from different sources can have different semantics for status checking and event notification.

To assumption 5: Any inter process or inter thread communication can be mapped to the process of delivering a message with the only relevant event being *message available*. Special parameters like priorities and deadlines can easily be embedded within the message.

With assumption 3 the scheduler can only assign work to threads and influence the selection process by assigning priorities. Assigning work to a thread is clearly an internal event. The structure of internal events is according to assumption 5 uniform and unlike external events. Scheduling for internal and external events is therefore realized in two different structures.

Internal event scheduling

Waiting for internal events happens in the scheduler. The scheduler can delay and reorder messages and adjust thread priorities on message delivery or on request. Each thread of a subtask registers itself when "entering" its processing block with the scheduler. The threads execution is suspended using appropriate mechanisms. On event arrival the scheduler selects the next appropriate and free thread or queues the event until such a thread becomes available. The message is delivered to the selected thread, its status is changed to busy and its execution is resumed. Events arriving at the same time are delivered when possible to different threads or queued. Delivery order is determined by event priority, execution order is determined by the system scheduler according to thread priorities and scheduling policies.

Each subtask is required to report successful execution of predefined processing blocks via checkpoints. A checkpoint allows the scheduler to readjust thread priorities but not to interrupt or delay execution. On task completion the thread reports back to the scheduler and is marked as free again.

External event scheduling

Special subtask wait for external events and map them to internal events. More that one external event can be processed in parallel. Each event originator type has is own waiting module. Waiting modules can have the following event check modes:

blocking the wait operation returns only when a new event was detected

non blocking the wait operation returns immediately and reports if a new events has been detected

blocking with timeout the wait operation returns when a new event has been detected or the timeout has been expired

Only one waiting module for each event originator type per thread is allowed. Waiting modules can only be combined in the same thread when they support polling in non blocking or blocking with timeout mode. Blocking waiting modules are not allowed in a single threaded environment.

Checkpoints

Checkpoints are used to inform the scheduler about the current state of processing. The caller has to provide an estimate about how far its work has progressed. The scheduler can use this estimate calculate if the current thread will meet its deadline and QoS requirements and try to adjust parameters accordingly when necessary. Most general purpose operation system allow only adjustments to scheduling priorities, some modern operating system also allow to specify special real time scheduling policies. It is the responsibility of the internal event scheduler to translate the application requirements into those values.

6.3 Design rules

MICO is currently a stable product. Changes and new features are integrated very carefully. Interested users have in the typical OpenSource fashion done independent code reviews and submitted bug-fixes. This has to lead a stable code base. But MICO has also accumulated ballast. The current CORBA 2.2 specification only prescribes the Portable Object Adaptor (POA), MICO still contains the Basic Object Adapter (BOA) for compatibility reasons. The original ORB design was optimized to fit the BOA's needs, little has been done to that original structure. Furthermore memory usage and access optimization have not been an issue before. Last, but most important, internal ORB structures have not been chosen and designed for application in an multi threaded environment.

Those considerations have lead to the following design and implementation principles when integrating multi threading into MICO:

step-by-step approach
: to minimize the risk of introducing new bugs, new features and functions are only implemented as needed and when the previous code base has proven to be stable

evolutionary process
: large changes to current behavior are broken down into small steps and each step is evaluated separately, this process is also known as minimal changes approach

functionality first
: the first priority is correct code behavior, changes to improve existing code or to optimize used algorithms are done only to stable versions

6.4 Evaluation

The prototype implementation has been tested with a number of simple benchmarks[4].

4 The benchmarks have not been evaluate for the correctness. The figures obtained should be considered as performance indicators only.

Benchmarks

A number of benchmarks have be performed. To get correct figures, only the time spend within the ORB has been measured. On the server side a timestamp has been attached to all invocation requests when they enter the ORB and has been evaluated when the requests answer leaves the ORB. On the client side a timestamp has been attached to the request when it is created and the total time needed is evaluated on return to the user application level. A second timestamp is used to calculate the time spend outside to current ORB's context and is subtracted from the total time. That way overhead incurred by the server and the message transport to the server is excluded.

The server side object is a dummy method returning immediately to the caller. Since the overhead incurred by a single call to an empty method is negligible small compared to overhead to calculate the time spend in this call, no attempt has been made to exclude the time spent in this call from the benchmark. All benchmarks have been performed with multi-threaded MICO ORB's.

The benchmark client performs invocations on the dummy server with an adjustable number of threads and number of invocations. The total time over all invocations (blue line) and the average time for a single invocation (red marks) is reported. All tests have been repeated 5 times. The minimum and maximum times during those runs are indicated in the diagrams with error-bars. The average time indicate in the diagram is the simple numeric average calculated by $max + min/2$. A simple approximation of the average time for a single invocation is given to show an overall trend (green line). Unless otherwise noted was the server configured to use a maximum of 10 threads per pipeline stage.

Number of invocations

Due to the nature of CORBA invocations, includes every measurement series a small overhead of time for connection establishment, object activation and other ORB internal housekeeping tasks. The precision of the used timers also has a certain impact on the results. To have reference of the impact of those things on the benchmark results a simple test has been performed with a variable number of invocations and 10 client threads. Figure 6.3 shows the result. For more than 100 invocation the measured time only differs in small amounts. For all following test a minimum number of 100 invocations has therefore been chosen.

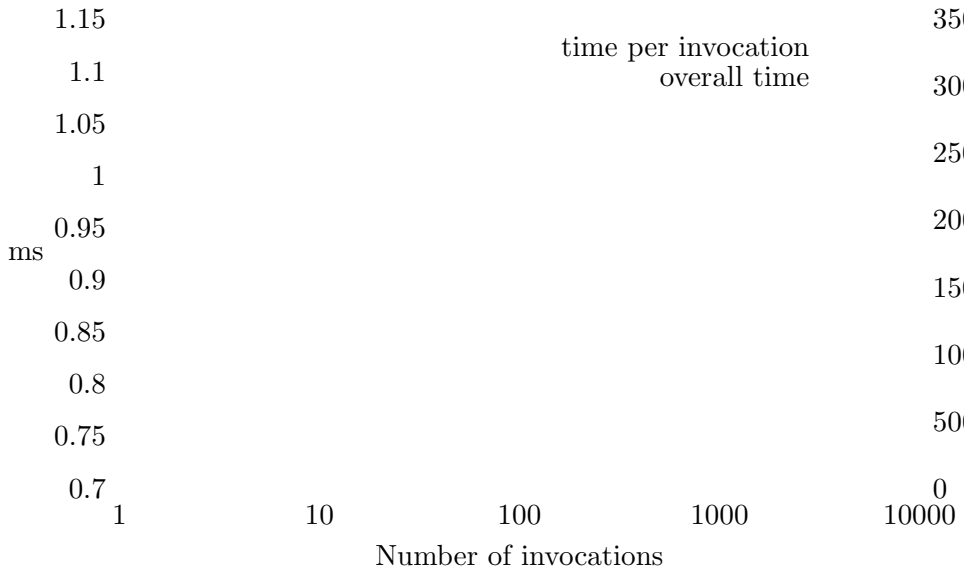


Figure 6.3: Relation between number of invocations and overall execution time

Single threaded client and server

As a reference has the average invocation for a single threaded client on a single threaded server been measured for 1000 invocations. The time spend in the client was 0.711ms and the time spend in the server was 0.974ms.

Client threads

Figure 6.4 shows the relation between the number of client threads (x-axis) and the duration for a single invocation (y-axis).

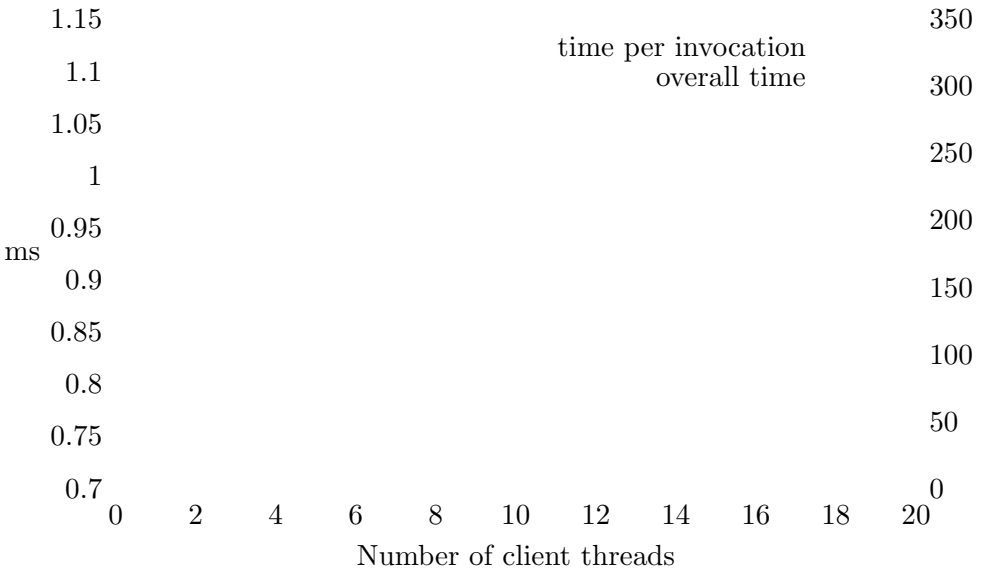


Figure 6.4: Relation between client threads and invocation time

The Diagram shows that the implementation becomes more effective with an increasing number of client threads. That indicates that the available processing resources are better used.

Server threads

Figure 6.5 shows the average invocation times on the server side in relation to the maximum threads per pipeline stage under different load situations.

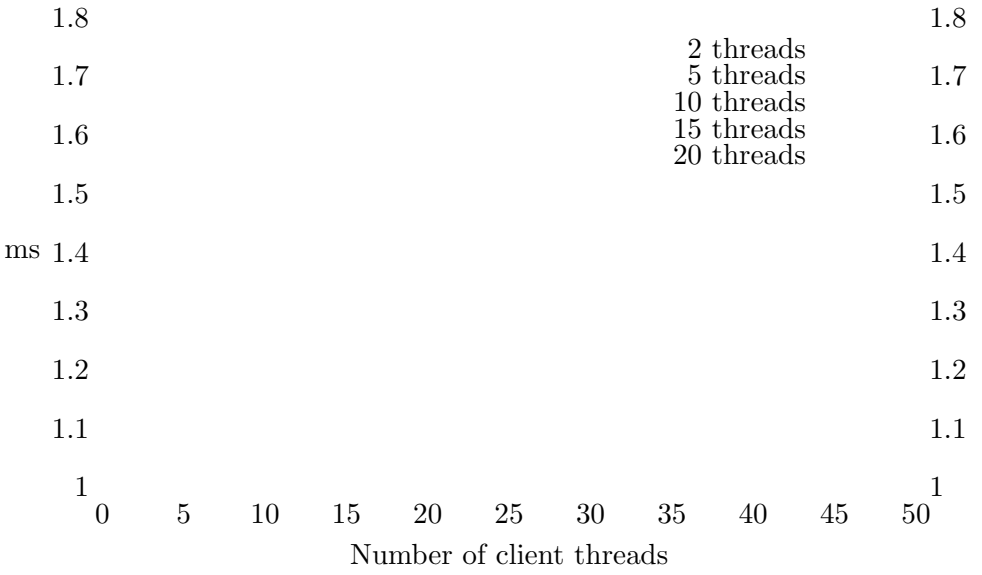


Figure 6.5: Relation between server threads and invocation time

Since this benchmark has been performed on a single CPU system is it not surprising that all curves are almost the same. After all is it not possible to use more that 100% CPU time. The fact that with increasing load the average invocation time decreases can be understand as an indication that no unnecessary wait situations exist.

Conclusions

The benchmark results clearly show that the multi threaded MICO ORB is able to make effective use of all available execution resources when enough demands exists. The longer delays when used with a low number of client threads indicate that more time is spend for handing the requests down the execution pipeline. That is not surprising considering the way how message passing is implemented. With low load on a pipeline stage all threads will be idle and waiting for new requests to arrive. A arriving request will cause a thread to become unblocked. Under higher load the likelihood of a request being assigned to a thread that attempts to switch its state from busy to idle increases. Such a thread will not be blocked but instead will immediately process the new request. This direct execution path is faster than unblocking blocked threads.

That leads to the direct conclusion that under low load situations it would be beneficial to bound requests to threads and have the same thread execute all pipeline stages.

6.5 Lessons learned and further work

Multi threading framework

Adapting existing object oriented and well structured code to the described framework has been proven to be possible. Only minimal changes to the main processing algorithms where necessary. The resulting ORB appears to be stable, have a good performance and scale well on high end systems, but further investigation into stability and high load behavior under production conditions is necessary. First experiences in a real life project indicate that CORBA compliant applications as well as applications using special features of the framework them self are being feasible and usable without any unexpected problems so far.

Prototype implementation

While adding multi threading capabilities to MICO some shortcomings and problems of the current implementation have been identified.

MICO's code base has grown extensively since its original implementation in 1998. Modules like the POA and several services have been added. Most work has been done for special purposes like evaluating specialized CORBA concepts or in the scope of master thesis' [14]. None of those projects had efficiency as a major goal and speed improvement features have been added only sporadic. All this has lead to a design that carries tons of old "cruft" like dead or inefficient code and non optimal interactions between components. Some of those problems had to be addressed like the ORB object-adapter interface (↗ Sect. 6.2.1) to guarantee basic multi threading functionality, others are still pending. Many of the additional services present

in MICO can benefit from multi threading capabilities but only the naming service has been changed to be thread safe.

The micro ORB approach provides a highly flexible interface to internal and external components. This flexibility guarantees easy extensibility but also creates the potential for highly sophisticated and obscured dependencies, interactions and side effects between components. These dependencies are often not documented and sometimes not even intended[5]. A significant amount of work has gone into investigating those. Some are documented in Sect. 6.1 others have been modified to behave like documented in [13].

5 At least according to the original MICO design document [13].

Acronyms

AMI Asynchronous Messaging Interface defined in [15]

API Application Programming Interface

BOA Basic Object Adapter

CORBA Common Request Broker Architecture is a standard for a middle-ware platform that allows applications to communicate with one another no matter where they are located or who has designed them. It includes among others the Interface Definition Language (IDL) and the Application Programming Interface (API) to access the ORB and its components (see also [7])

DCOM Distributed Component Object Model is a object oriented middle-ware implementation developed by Microsoft Corp. and used mainly on the Microsoft Windows platform.

DII Dynamic Invocation Interface

GIOP General Inter-ORB Protocol on the wire communication protocol used between ORB's

IDL Interface Definition Language describes the interface that client objects call and object implementations provide.

IMR Implementation Repository

IIOP Internet Inter-ORB Protocol Protocol used to map GIOP onto the TCP/IP protocol.

IR Interface Repository

OA Object Adapter

OMG Object Management Group is promoting the development of component based software. They have defined standards such as CORBA and UML (see also [3])

ORB Object Request Broker is responsible for all of the mechanisms required to find the object implementation for a request, to prepare the object implementation and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object's interface.

POA Portable Object Adaptor

QoS Quality of Service

RFP Request For Participation The Object Management Group (OMG)'s way to invite its members to comment on newly proposed standards.

RMI Remote Method Invocations

SII Static Invocation Interface

UML Unified Modeling Language^{TM}

Bibliography

[1] Hans Arno Jacobsen and Boris Weissman. Towards High-Performance Multithreaded CORBA Servers. Proceedings of The International Conference on Parallel and Distributed Processing Techniques and Applications {PDPTA '98}.

[2] DARPA Quorum project. http://www.darpa.mil/ito/research/quorum/.

[3] OMG. The Object Management Group. http://www.omg.org/, 1989.

[4] OMG. What is CORBA? http://www.omg.org/corba/whatiscorba.html, 2000.

[5] International Organization for Standardization. Open Systems Interconnection – Basic Reference Model: The Basic Model, ISO/IEC 7498-1:1994.

[6] Jerome H. Saltzer, David P. Reed, and David D. Clark. Ent-to-End Arguments ins System Design. *Second International Conference on Distributed Computing Systems*, pages 509–512, April 1981.

[7] OMG. formal/99-10-07: CORBA/IIOP 2.3.1 Specification. http://www.omg.org/, The Object Management Group, 1999.

[8] Emery D. Berger and Robert D. Blumofe. Hoard: A Fast, Scalable, and Memory-Efficient Allocator for Shared-Memory Multiprocessors. http://www.hoard.org/, Department of Computer Sciences, The University of Texas at Austin, 1999.

[9] Douglas C. Schmidt. Strategized Locking, Thread-safe Interface and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components. *C++ Report, SIGS*, Vol. 11(No. 9), September 1999.

[10] A. Puder, K. Römer, F. Pilhofer. MICO – Mico Is Corba. http://www.mico.org/.

[11] Open Source: Software Gets Honest. http://www.opensource.org/.

[12] The Open Group. http://www.opengroup.org/.

[13] K. Römer. *MICO* – MICO is CORBA – Eine erweiterbare CORBA–Implementierung für Forschung und Ausbildung. Diplomarbeit, J. W. Goethe–Universität Frankfurt/Main, 1998.

[14] F. Pilhofer. Design and Implementation of the Portable Object Adapter. Diplomarbeit, J. W. Goethe–Universität Frankfurt/Main, 1999.

[15] OMG. orbos/98-05-05: Asynchronous Messaging. http://www.omg.org/, The Object Management Group, 1998.