



Pattern–Oriented Software Architecture for Concurrent and Networked Applications

Master thesis

Papacharisiou Konstantinos

Supervisor:

Dimitrios Katsaros, Lecturer

Committee members:

Panayiotis Bozanis, Associate Professor

Catherine Housti, Professor



Αρχιτεκτονική Λογισμικού με Επαναχρησιμοποιήσιμο Κώδικα στην Ανάπτυξη Δικτυακής Εφαρμογής

Μεταπτυχιακή Διατριβή

Παπαχαρισίου Κωνσταντίνος

Επιβλέπων:

Δημήτριος Κατσαρός, Λέκτορας

Μέλη επιτροπής:

Παναγιώτης Μποζάνης, Αναπληρωτής Καθηγητής

Αικατερίνη Χούστη, Καθηγήτρια

Contents

Abstract	7
Περίληψη	8
Chapter 1 Introduction	11
1.1. Definition and pattern categories	11
1.2. Importance of Concurrent and Networked patterns	11
Chapter 2 Proxy pattern	13
2.1. Motivation	13
2.2. Solution	13
2.2.1. An example in C++	14
Chapter 3 The Active Object pattern	16
3.1. Motivation	16
3.2. Problem	16
3.3. Solution	16
3.3.1. Dynamics	18
3.4. Using object-oriented C++ features	20
3.4.1. Using runtime binding	21
3.4.2. Polymorphic future	22
3.5. Benefits and limitations	23
Chapter 4 Scoped Locking pattern	24
4.1. Motivation	24
4.2. Problem	24
4.3. Solution	25
4.4. Benefits and limitations	26
Chapter 5 Half sync – Half async pattern	27
5.1. Motivation	27
5.2. Problem	27
5.3. Solution	28
5.3.1. Dynamics	29
5.4. Benefits and limitations	31
Chapter 6 Leader/Followers	33
6.1. Motivation	33
6.2. Problem	33

6.3.	Solution	34
6.3.1.	Dynamics	34
6.4.	Benefits and limitations	36
Chapter 7	Developing a real server environment using patterns	38
7.1.	Functionality	38
7.2.	UML Design diagram	40
7.3.	UML Class components diagrams	41
7.3.1.	Communication interface	41
7.3.2.	Invocation – execution method decoupling	42
7.3.3.	Sync/async management & parallelism escalation	43
7.4.	Concurrency management	45
Chapter 8	Conclusion and future work	46
8.1.	About the patterns	46
8.2.	About the application	46
Appendix		47
	Scheduler	47
	Client	49
	Servant	51
	Method Request	53
	Proxy	55
	Activation Queue	57
	PUT & GET	59
References		60

Dedicated to my family and my friends

Abstract

Object Oriented (OO) programming generally incorporates some exciting features able to give not only flexible and effective but also structured and reusable solutions. The discipline of Distributed and Networked applications can reveal the power of such OO languages applied in an effective manner to a variety of problems.

Using this exciting tool (OO languages) of software engineering patterns we are going to examine how applying C++ can solve some of the most common algorithmic problems in distributed and networked systems. Such solutions have appeared, applied and generalized in Distributed and Networked systems since 1990. As a result some patterns have emerged and optimized regarding the most common of the challenges that programmers have struggled with.

The intention of this thesis is to represent, study and evaluate the features of the most common patterns appeared in Distributed and Networked applications.

Περίληψη

Ο Αντικειμενοστραφής Προγραμματισμός (ΑΠ) ενσωματώνει ορισμένα εκπληκτικά χαρακτηριστικά που μπορούν να δώσουν αποτελεσματικές, ευέλικτες καθώς επίσης σωστά δομημένες και επαναχρησιμοποιήσιμες λύσεις. Ο τομέας των Κατανεμημένων και Διαδικτυακών εφαρμογών μπορεί να αποκαλύψει την δύναμη μίας τέτοιας γλώσσας που χρησιμοποιείται αποδοτικά για τη λύση μίας πληθώρας προβλημάτων.

Χρησιμοποιώντας αυτό το εκπληκτικό εργαλείο (ΑΠ) της επιστήμης των υπολογιστών θα εξετάσουμε πως εφαρμόζοντας τη C++ μπορούμε να λύσουμε μερικά από το πιο κοινά προβλήματα που προκύπτουν στα Κατανεμημένα και Διαδικτυακά συστήματα χρησιμοποιώντας πρότυπα. Τέτοιες λύσεις εφαρμόζονται, επεκτείνονται, γενικεύονται και βελτιστοποιούνται σε Κατανεμημένες και Διαδικτυακές εφαρμογές από το 1990. Σαν αποτέλεσμα έχουν προκύψει κάποια πρότυπα λύσεων στις πιο κοινές προκλήσεις που έχουν αντιμετωπίσει οι προγραμματιστές.

Σκοπός της παρούσας εργασίας είναι η μελέτη, η παρουσίαση και η αξιολόγηση των πιο κοινών προτύπων που εμφανίζονται στις Κατανεμημένες και Διαδικτυακές εφαρμογές.

List of Figures

Picture 1 Proxy pattern example.....	13
Picture 2 Simple Proxy Algorithm.....	14
Picture 3 Output of Proxy Pattern Example	15
Picture 4 Active Object Pattern Structure	17
Picture 5 Active Object Operations	18
Picture 6 Active Object UML	20
Picture 7 Active Object Dispatch function Implementation.....	21
Picture 8 Active Object Future function Implementation	22
Picture 9 Locking Problem Scenario	24
Picture 10 Scoped Locking function	25
Picture 11 Scoped Locking Implementation.....	25
Picture 12 Scoped Locking use	26
Picture 13 Half sync – Half async necessity	28
Picture 14 Half sync – Half async Structure.....	28
Picture 15 Half sync – Half async Dynamics	30
Picture 16 Half sync – Half async utilization in ACE framework.....	31
Picture 17 Half sync-Half async weaknesses	33
Picture 18 Leader/Followers UML.....	34
Picture 19 Leader/Followers Dynamics.....	35
Picture 20 The Server Environment	38
Picture 21 Server Features	39
Picture 22 Server Design Diagram	40
Picture 23 Communication Interface Class Diagram	41
Picture 24 Method Decoupling Class Diagram	42
Picture 25 Sync/Async Management & Parallelism Class Diagram	43
Picture 26 Running Instance.....	44
Picture 27 Concurrency Management Dynamics.....	45

List of Tables

Table 1 Active Object context	16
Table 2 Scoped Locking Context.....	24
Table 3 Half sync – Half async context	27
Table 4 Leader/Followers Context	33

Chapter 1

Introduction

1.1. Definition and pattern categories

In software engineering, a **design pattern** [9] is a general **reusable** solution to a commonly occurring problem within a given context in software design. A pattern is **not a finished design** that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

Patterns are formalized best practices that the programmer must implement in the application. **Object-oriented design patterns** typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply object-orientation or more generally mutable state are not as applicable in functional programming languages.

Design patterns reside in the domain of modules and interconnections. There are many types of design patterns, for instance:

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to exploit application characteristics on a computing platform.
- **Computational design patterns** addressing concerns related to key computation identification
- **Execution patterns** that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization
- **Implementation strategy patterns** addressing concerns related to implementing source code
- **Structural design patterns** addressing concerns related to high-level structures of applications being developed

1.2. Importance of Concurrent and Networked patterns

When objects are distributed, the various entities that constitute these objects must communicate and coordinate with each other effectively. Moreover, they must continue to do so as applications change over their lifetimes. The placement of objects, the available networking infrastructure, and platform concurrency options allow for a level of freedom that is powerful, yet challenging.

When designed properly, concurrent object-oriented network programming capabilities can add a great deal of flexibility to your application options. For instance, in accordance with the requirements and resources available to your projects, you can use

- Real-time, embedded, or handheld systems
- Smartphones, personal or laptop computers

- and even supercomputers

These are the fields that patterns have contributed the most as reusable solutions and have boosted programming effectiveness. Concurrent patterns are a family of interrelated patterns that define a process for systematically resolving problems that arise when developing software for distributed systems.

Chapter 2

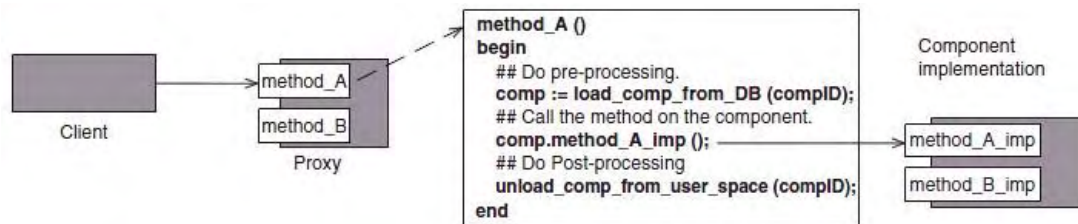
Proxy pattern

2.1. Motivation

Software systems consist of cooperating components: client components access and use the services provided by other components. It is often impractical, or even impossible, to access the services of a component **directly**, for example because we must first check the access **rights** of its clients, or because its **implementation** resides on a remote server. If clients are to access a remote component directly, they become dependent on the component's location, as well as on the networking protocols that are used to access its functionality, which should be transparent to a component's clients.

2.2. Solution

So a solution could be the encapsulation of the components functionality in a **separate** surrogate of the component- the **proxy**- and let the clients to communicate only through the proxy rather than the component itself.



Picture 1 Proxy pattern example

A **proxy** frees both the client and the component from implementing component-specific housekeeping functionality. It is also transparent to clients whether they are connected with the component or its proxy, because both publish an identical interface. The primary liabilities of a proxy are the hidden costs it can introduce for clients, although for many uses these costs are negligible compared to the execution time of the component's services.

Some common **types** of **proxy patterns** are:

- **Client Proxy** shields the clients of a remote component from network addresses and IPC protocols to enable location independence within a distributed system: clients can use the client proxy as if it were a local component.
- **Business delegate** goes one step further: it shields clients from all IPC, as well as locating remote components, load balancing when multiple component instances are available in a distributed application, and handling of specific networking errors.
- **Threadsafe interface** is a proxy that serializes access to concurrent components transparently for both the client and the components.

- **Counting handle** is normally expressed as a proxy that helps to access the functionality of shared heap objects whose lifetime must be managed explicitly by an application, to avoid memory leaks when the object is no longer used.

2.2.1. An example in C++

In the following example a proxy pattern version is presented. Not only does this illustration of the proxy pattern give an abstract access method to clients but also it **prevents from multiple loadings** leading to a memory overflow as we can see in output.

```
#include <iostream>

using namespace std;

//on System A
class server{
private:
    int type;
public:
    server(int tp):type(tp){}

    void downloadImage(){
        cout << "downloading..." << "type: " << type << endl;
    }
};

//on System B
class proxy{
private:
    server *s;
    int type;
public:
    proxy(int tp): type(tp), s(NULL) {}

    void displayImage(){
        if (s==NULL){
            s = new server(type);
            s->downloadImage();
        }
        else
            cout << "omitting call..." << endl;
    }
};

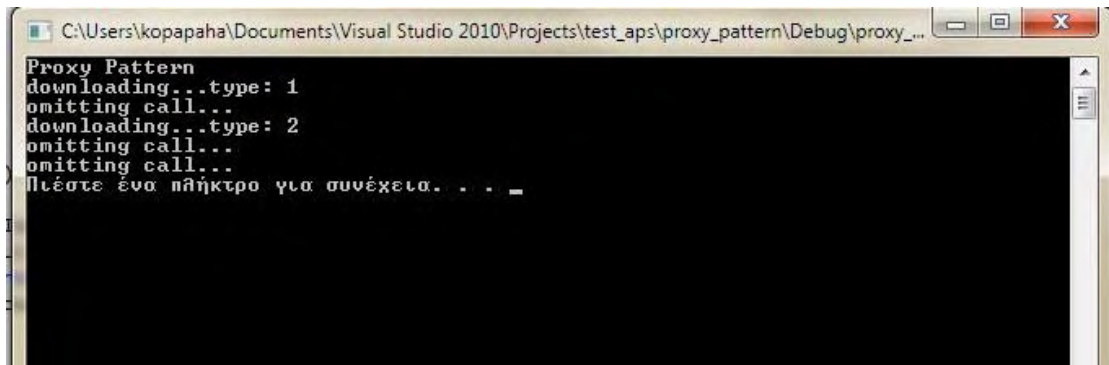
int main(){

    cout << "Proxy Pattern" << endl;

    proxy p1(1);
    proxy p2(2);

    p1.displayImage();
    p1.displayImage();
    p2.displayImage();
    p1.displayImage();
    p2.displayImage();
    system("PAUSE");
    return 0;
}
```

Picture 2 Simple Proxy Algorithm



```
C:\Users\kopapaha\Documents\Visual Studio 2010\Projects\test_aps\proxy_pattern\Debug\proxy_...
Proxy Pattern
downloading...type: 1
omitting call...
downloading...type: 2
omitting call...
omitting call...
Πιέστε ένα πλήκτρο για συνέχεια. . . _
```

Picture 3 Output of Proxy Pattern Example

Chapter 3

The Active Object pattern

3.1. Motivation



A web server must be able to escalate efficiently as # of clients grows



If we have hundreds or thousands of users in a big crowd with their smart phones, or laptops, or other computing devices, all pounding away at our server, then we're simply not going to be able to get any kind of response time if there's one thread of control. So we clearly need a way to **leverage the available hardware** and software much more effectively.

3.2. Problem

Context	Problem
High – performance web servers that need to leverage advances in hardware and software	To improve Quality Of Service for all connected clients, a web server must not block while waiting for connection flow control to abate & must fully leverage parallelism

Table 1 Active Object context

To do that, we're going to apply the **active object pattern**. And we're going to use this active object pattern to scale up the service performance by processing the various requests, in separate threads of control. In particular we're going to have a **separate thread** of control for **each connection** that comes in from a client.

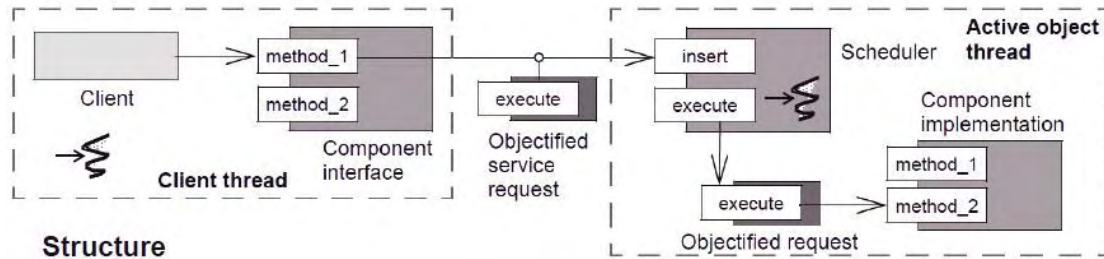
In general the active object pattern defines the units of concurrency to be service requests that are run on a component. And it arranges to have these requests for service work in a manner where the method that requests the service is **decoupled from** the method that executes the service so those can run concurrently.

3.3. Solution

The Active Object Pattern **decouples** the method **execution** from method **invocation** in order to simplify synchronized access to an object that resides to its own thread of control. The pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer- consumer and reader/writer

applications are relevant to this model of concurrency. As Douglas C. Schmidt [5] mentions this pattern is commonly used in distributed systems requiring multi-threaded servers.

We're going to start by analyzing the **structural dimensions** at first. One part of the structure involves the client that runs in a thread of control and invokes a method on the active object. This method is actually invoked on a **proxy**, and what happens is that proxy converts that method call into a message called a **method request**.

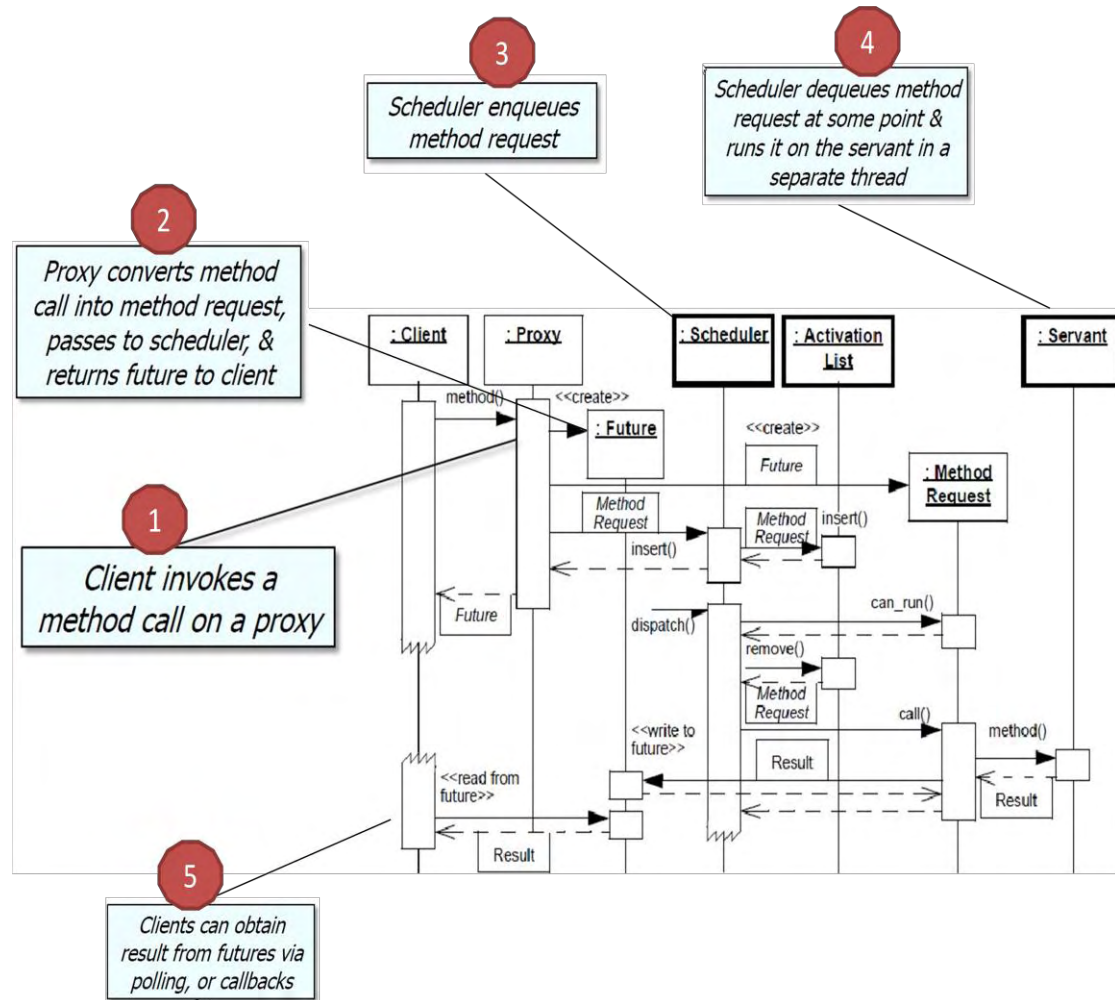


Picture 4 Active Object Pattern Structure

That method request is passed over to the **active object** where it's in queued in an activation list (**activation queue**) that the active objects scheduler manages. At some later point in time that **scheduler**, running in a separate thread of control from the client, we'll DQ the message and execute the task it describes.

3.3.1. Dynamics

Here's a dynamic view that gives you a little bit more perspective of what's happening as these various interactions take place between the various roles in the pattern.



Picture 5 Active Object Operations

We start out by having a client invoke a method call on a **proxy** (we can use the proxy pattern). In our particular case, the proxy will convert the method call into a message called a method request and this method request basically contains message fields it corresponds to the data, it keeps track of the name of the method. They also have some return information as well, and it gets bundled up into an object that you can use to pass from the proxy back into the active object portion. The proxy part also **creates the future**, which is going to be used to pass back to the client later, so it can subsequently retrieve the results, if any that occur when the active object invokes this request.

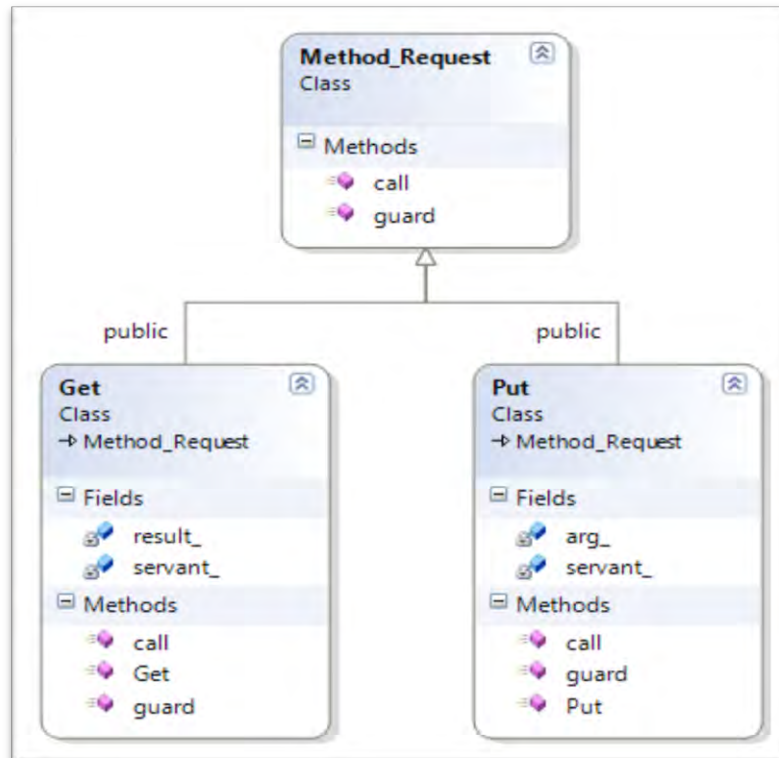
Afterwards the method request is then passed to the **scheduler**. The scheduler enqueue this method request on an activation list and it gives back the future to the client. In later time, which is depended from a variety of different factors, depends on scheduling constraints that the scheduler may know about or on certain guard conditions that the method request knows about, scheduler may simply run things in FIFO order, depending on how everything

is implemented. The schedule will then queue the request and then later pull it off, convert it back into a method call on to the servant that actually carries out the request. And after the servant is done, the servant will end up, as a side effect, **updating** the **future** so the client can then get the result.

There's a couple of ways in which **clients** can get results from futures. One way they can do it, is they can periodically poll the future and ask if the result is ready. Or they might do a timed poll where they'll say, I'll wait up to a couple seconds for the result, and then I'll go off and do some other things. An alternative model, which is probably more scalable in many ways, is to use **callbacks**. So what happens here is, when the future's value is updated, when the server has finished its processing, then a callback takes place to deliver those results back to the client.

3.4. Using object-oriented C++ features

Implementing the active object pattern some key features of C++ are emerging.



Picture 6 Active Object UML

Each method of a **Proxy** transforms its invocation into a **Method Request** and passes the request to its **Scheduler**, which enqueues it for subsequent activation. A Method Request base class defines virtual `guard` and `call` methods that are used by its Scheduler to determine if a Method Request can be executed and to execute the Method Request on its Servant.

The methods in this class must be defined by **subclasses**, one subclass for each method defined in the Proxy. The rationale for defining these two methods is to provide Schedulers with a **uniform interface** to evaluate and execute concrete Method Requests. Thus, Schedulers can be decoupled from specific knowledge of how to evaluate the synchronization constraints or trigger the execution of concrete Method Requests. For instance, when a client invokes the `put` method on the Proxy in our example, this method is transformed into an instance of the `Put` subclass, which inherits from `Method Request` and contains a pointer to the Servant.

3.4.1. Using runtime binding

The next code fragment presents the `dispatch()` method of `scheduler`. We can understand the importance of **virtual methods guard()** and **call()** of class `Method_Request`. Since `guard()` and `call()` are virtual methods we have dynamic binding (dereference at runtime) in each invocation of them. Consequently, the correct Servant method is called in order to fulfill the request.

In further details, the implementation of `dispatch` method is kind of **abstract** as the **guard** and **call** methods must implementing different restriction policies regarding the type of method request residing in the queue (**PUT** or **GET**). So this differentiation in the code structure enhances the maintenance effort and programming development.

```
virtual void
MQ_Scheduler::dispatch (void)
{
    // Iterate continuously in a
    // separate thread.
    for (;;) {
        Activation_Queue::iterator i;
        // The iterator's <begin> call blocks
        // when the <Activation_Queue> is empty.
        for (i = act_queue_>begin (); i != act_queue_>end ();i++) {
            // Select a Method Request 'mr'
            // whose guard evaluates to true.
            Method_Request *mr = *i;
            if (mr->guard ()) {
                // Remove <mr> from the queue first
                // in case <call> throws an exception.
                act_queue_>dequeue (mr);
                mr->call ();
                delete mr;
            }
        }
    }
}
```

Picture 7 Active Object Dispatch function Implementation

3.4.2. Polymorphic future

Likewise the use of **operators overload** enhances the implementation of message future requests. As Schmidt mentions [1] future requests can be evaluated in two ways. **Immediate evaluation** blocks the calling thread until result is ready. Otherwise in **deferred evaluation** we can obtain a future object without thread blocking and evaluate result in future.

```
template <class T>
class Future
{
    // This class implements a 'single write, multiple
    // read' pattern that can be used to return results
    // from asynchronous method invocations.
public:
    // Constructor.
    Future (void);
    // Copy constructor that binds <this> and <r> to
    // the same <Future> representation
    Future (const Future<T> &r);
    // Destructor.
    ~Future (void);
    // Assignment operator that binds <this> and
    // <r> to the same <Future>.
    void operator = (const Future<T> &r);
    // Cancel a <Future>. Put the future into its
    // initial state. Returns 0 on success and -1
    // on failure.
    int cancel (void);
    // Type conversion, which obtains the result
    // of the asynchronous method invocation.
    // Will block forever until the result is
    // obtained.
    operator T ();
    // Check if the result is available.
    int ready (void);
private:
    Future_Rep<T> *future_rep_;
    // Future representation implemented using
    // the Counted Pointer idiom.
};
```

Picture 8 Active Object Future function Implementation

3.5. Benefits and limitations

Some benefits of this pattern are:

- **Enhances concurrency & simplifies synchronized complexity:** Client threads and asynchronous method executions can run concurrently. A scheduler can evaluate synchronization constraints to serialize access to servants.
- **Transparent leveraging of available parallelism:** Multiple active object methods can execute in parallel if supported by the OS/hardware
- **Method execution order can differ from method invocation order:** Methods invoked asynchronously can be executed according to synchronization constraints defined by guards and scheduling policies

Some limitations this pattern may have:

- **Higher overhead:** Depending on how an active object is implemented, context switching, synchronization, & data movement/ copying overhead may occur when invoking, scheduling, & executing active object method calls
- **Complicated debugging:** It is harder to debug programs that use concurrency due to non-determinism of the various schedulers.

Chapter 4

Scoped Locking pattern

4.1. Motivation

Another dimension of synchronization is to ensure that locks are **released properly**. In single-threaded code, releasing locks is not an issue. When you begin to write multi threaded code, of course, releasing locks is a big deal. If locks not be released properly, you could end up with dead lock, or other kinds of inefficiencies in your solution.

This is a particularly tricky thing because it's not always clear whether we're locking things for real or not. We might have null locks or recursive locks or non-recursive locks. But in any case, we have to write the code that **acquires** a lock on the entry to a critical section and **releases** it on the way out. If we're not careful and we don't release the lock, likely we will end up to a critical section problem.


4.2. Problem

Context	Problem
A shared resource that is concurrently accessed & updated b multiple threads must be protected by a lock	If programmers acquire & release locks explicitly it's hard to ensure that locks are released in all code paths

Table 2 Scoped Locking Context

Here is an example from a message queue class implementation, where a lock that has been acquired is not released properly due to its runtime execution path:

```
template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
public:
...
    int dequeue_head (ACE_Message_Block &*mb, ACE_Time_Value *to) {
        lock_.acquire ();
        ...
        lock_.release ();
    }
...
}
```

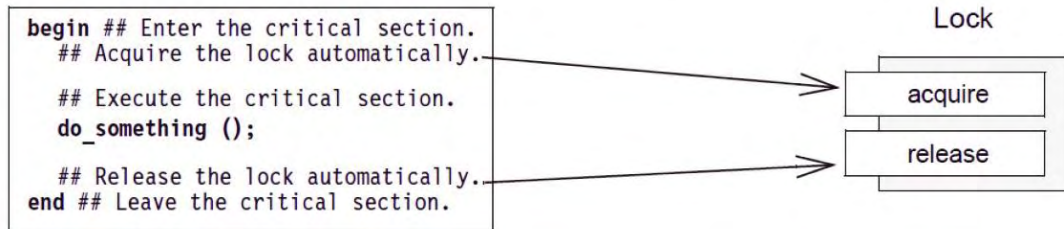


Control can leave scope in C++ prematurely due to return, break, continue, goto, or unhandled exception

Picture 9 Locking Problem Scenario

4.3. Solution

We're going to apply the **Scoped Locking** pattern. This pattern will make it possible to use some **features of object oriented** languages, where constructors acquire resources, and destructors release them, to ensure that the scope over which things are applied, will always properly acquire and release the locks **automatically**.



Picture 10 Scoped Locking function

For example let's see how we might apply scope blocking to the ACE [9] Message Queue implementation. What we're going to do is we're going to define a **guard class**. This guard class, which is a **template class**, follows a very simple protocol. Its **constructor** is going to acquire a lock that corresponds to the parameter that's passed into the template. And its **destructor** is going to release that lock. So when we come in, the constructor acquires a lock and the destructor releases the lock. Thereby ensuring that we always are going to release the lock, whether or not we have an exception or a return or a goto or anything else that will allow that **scope** to be exited. The ACE guard which here act as a helper class also illustrates another reason why wrapper facades are so powerful. So, all we have to be able to do to use all these different mechanisms is to create a parameterized guard class which is a helper class that implements the scoped locking pattern. Then we can parameterize it with a particular type of the wrapper facade.

1. Define a guard class that acquires & releases a particular type of lock in its constructor & destructor, respectively

```

template <typename LOCK>
class ACE_Guard {
public:
  ACE_Guard (LOCK &lock): lock_ (&lock) { lock_ ->acquire (); }
  ~ACE_Guard () { lock_ ->release (); }
  ...
private:
  LOCK *lock_;
}

```

Store a pointer to lock & acquire lock in constructor

Destructor releases lock when guard goes out of scope

Pointer to the lock that's being managed

Picture 11 Scoped Locking Implementation

The next thing we do is we let the critical sections correspond to the scope and lifetime of these guard objects. Here's an illustration using the dequeue head method that we had in the message queue from before. Now what we do when we come in is we use the scope locking pattern in order to be able to **automatically acquire the lock** in its constructor. And then no-matter how we leave the scope, short of crashing the program, that lock is going to automatically be released, which is much more powerful and much more fool proof. The mistakes are going to be eliminated because it automatically cleans things up.

2. Let critical sections correspond to the scope & lifetime of a guard object

```

template <typename SYNCH_STRATEGY>
class ACE_Message_Queue {
public:
...
    int dequeue_head (ACE_Message_Block *&mb, ACE_Timeout *to) {
        ACE_Guard<typename SYNCH_STRATEGY::MUTEX> guard (lock_);
        ...
    }
...

```

 Use Scoped Locking to acquire & release lock_ automatically

 lock_ released automatically when guard goes out of scope

Picture 12 Scoped Locking use

4.4. Benefits and limitations

There are a number of benefits of course to using this particular pattern:

- **Increased robustness:** This pattern increases the robustness of concurrent applications by eliminating common programming errors related to synchronization & multi-threading. It makes things more robust because the programmer can eliminate this common source of overheads where you have the mistake of forgetting to release the locks or jumping out of this scope too quickly and having problems. This of course is really an **embodiment** of a broader C++ idiom called **resource acquisition is initialization or RAII**.

There are some limitations using this particular pattern:

- **Potential for deadlock when used recursively:** One of the problems is that you can end up with dead lock if you use this recursively.
- **Limitations with language-specific semantics:** There is also some language semantics issues. We're using C++ programming language features. And while that will work for certain things, it doesn't know about low level features that are available in the operating system to do things like exit threads by thread exit calls.

Chapter 5

Half sync – Half async pattern

5.1. Motivation

Let's introduce more reasons about the motivation for why we want to improve upon **thread per-connection**. Thread-per-connection is certainly an improvement from a performance point of view, on a purely reactive solution that we had before. And it's also going to have lighter weight, less overhead characteristics relative to the, the activator based process per request or process per connection model.

But there are some downsides. If you have an environment where you've got large numbers of clients, and you may have very bursty data traffic, you can end up in situations where you've got hundreds or thousands of threads being spawned to handle the various connections. And if you only have a limited number of cores at your disposal, four, eight, twelve, having thousands of threads running around really doesn't give you much of a boost.

In fact, it's probably going to take a lot of **extra memory**. It's going to end up requiring a lot of extra work by the operating system to manage. All those internal thread queues and keep track of all the resources.

5.2. Problem

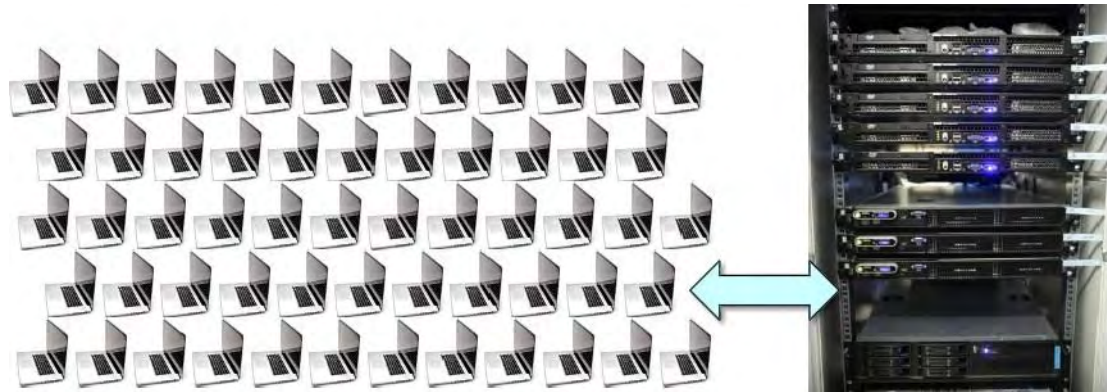
Context	Problem
Web servers that run in settings with large number of clients with burst request patterns	Allocating an OS thread per connected client does not scale without devoting enormous processing capability & memory to the servers

Table 3 Half sync – Half async context

So what we'd like to do is find a way to be able to **get concurrency but perhaps without paying quite so much overhead**, especially in ways that really doesn't help advance the performance of our system. So to do this, we're going to apply the Half-Sync/Half-Async pattern.

For example we can use this pattern in order to be able run our HTTP get request processing in separate threads of the control and to bound the number of threads that we allocate to keep them closer to the number of cores that we may have at our disposal.

In general, the Half-Sync/Half-Async pattern helps to decouple asynchronous and synchronous service processing in current systems.

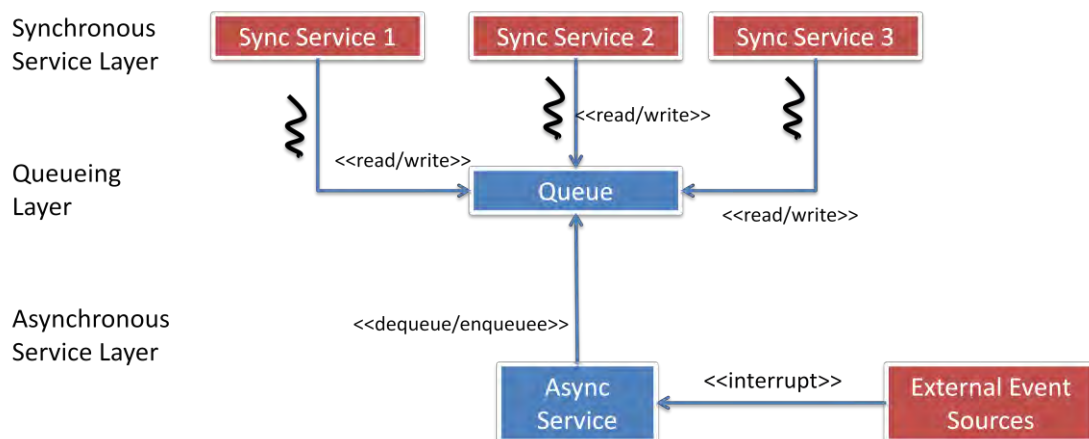


Picture 13 Half sync – Half async necessity

This pattern is often applied historically in operating systems. People often implement operating system kernels using this pattern. The kernel processing is typically driven by asynchronous callbacks, at which don't block. There's often not more than one or a handful of threads of control in the OS kernel to keep things efficient. In application process level those are able to **block**. You can block on **read calls**, you can block on **write calls**, and so on. In between those things, there's a queuing layer that's used to mediate interactions between the interrupt driven asynchronous kernel, and the more synchronous process in the user space. The sockets layer is a good example of such a queuing layer.

This pattern is also used in many modern user interface toolkits. For example, in **Android**, you typically do your user interface processing in the main thread, in one thread of control, and you can't afford to block that thread for more than a very short period of time. So if you want to do **longer running** operations you **spawn threads** and you run those as background processing and then you have the main thread coordinate with those other threads. These examples illustrate the core structural elements of this pattern.

5.3. Solution



Picture 14 Half sync – Half async Structure

Describing the structural fundamentals, at the bottom of this is the **Async** service layer. This is driven by incoming events. It could be driven by call backs, it could be driven by signal responses, it could be driven by interrupts, etcetera. The characteristic of this layer is that there's **only one thread of control**, one stack and so you can't afford to block for any length of time or you'll starve out all the other sources of events that happen to reside at this layer.

On top of the, the Async approach but beneath the next layer up is the **queuing layer**. When the Async layer is done processing the incoming events to the point where it might have to block to do anything further, it sticks the request onto a queue in the queuing layer.

That queuing layer is then used to service the various threads or processes that run in the **synchronous service layer**. And this is the layer where you can have multiple blocking threads or processes that sit there waiting for work on the queue. When the work comes in, then they can go ahead and do their own thing without worrying about perturbing or interfering with other things taking place in other threads or other processes. So that's the basic structure relationships between the various layers in this pattern.

5.3.1. Dynamics

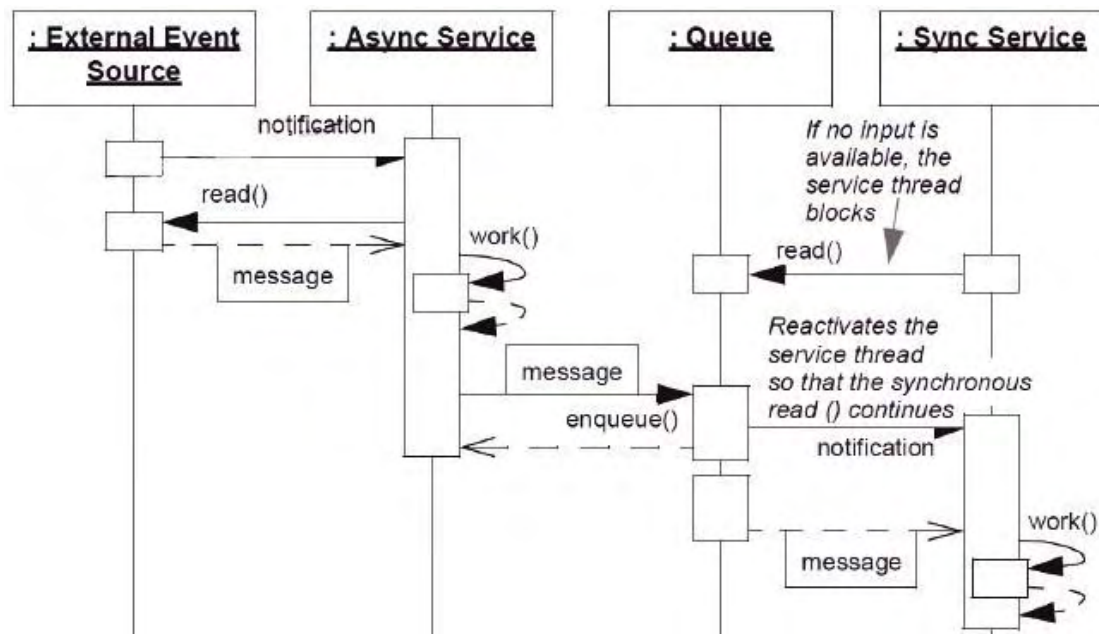
The dynamics of this pattern will help to explain some of the interactions between the different layers. So in this approach, the **Synchronous** services can run concurrently in multiple threads of control or processes that can block as well as to the processing that's taking place in the **Async** layer. So, typically you might do the event handling logic down in the Async layer and that won't block for any length of time.

When that is done processing an incoming request, it will typical enqueue that onto the message queue in the **queuing layer**. Afterwards one or more threads or processes will wait on that queue, pull the request off and then run to completion at their leisure without concern for weather they're stealing or **starving**.

Anybody else out because they have their own threads, they have their own independent instruction streams and their own resources and stacks and so on.

In an example about **how** we can **apply the pattern** let's first assume we have some bounded number of cores at our disposal. So, we would allocate some bounded number of threads that would be maybe a factor of two larger than the number of cores. Maybe having some **extra threads** is a good approach because if one thread is **blocked on I/O**, other threads can continue to run. What we're trying to avoid here is an unbounded number of threads. A thread per request or a thread per connection, could get unwieldy in a very dynamic environment.

Each thread in the Synchronous layer is now able to block, either because of flow control or it's doing some kind of long running right operation, where it has to read from some data or it's grabbed a lock or whatever it is that causes it to block, without having to worry about interfering with the quality of service and performance of the other threads at the Synchronous layer.



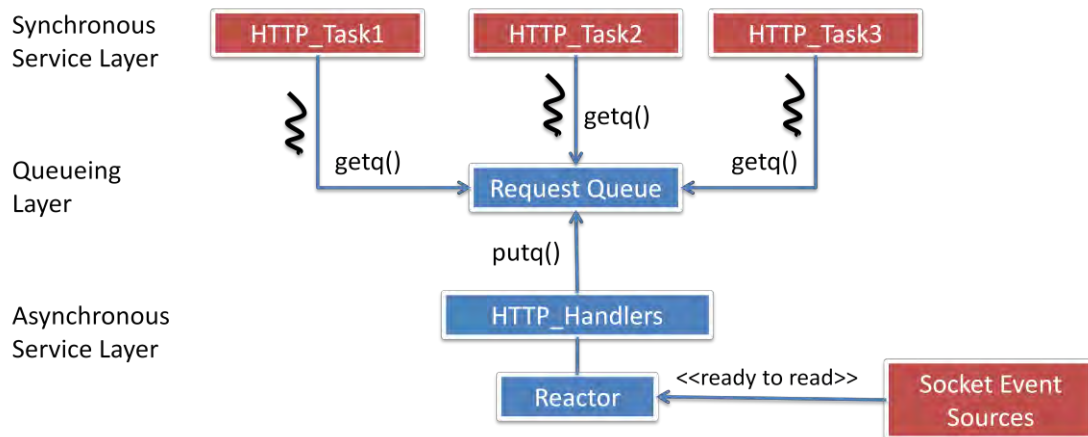
Picture 15 Half sync – Half async Dynamics

We can then use the Half-Sync/Half-Async pattern to implement a web server by combining the **proxy pattern** with the **active object** pattern. Here's the way we could do this. We could have a **proxy** at the bottom in the **Async layer** that's waiting for work to show up on multiple sources of input, multiple sources of socket events. When that work comes in, it will be processed by one of our (e.g. HTTP) service handlers, which will read the requests out of the buffer. When we finally got all the pieces in a get request we can then turn around and stick them onto a queue. That **queue** will then be used to service the various threads that are running in the **active object** layer. So you can think of the queue and the threads as being part of an active object, and the proxy as being used to feed that particular queue.

Then the threads that are running the object layer can dequeue the get-request, do the processing, and afford to block. The synchronized request queue, of course, is what mediates the interaction between the proxy portion and the active portion. So we have one thread in the **proxy** part, multiple threads in the **active object** part, and the queue is part of the active object that's used to feed those threads in that pool.

If for some reason, flow control occurs in one of the threads, we're able to keep making progress and other threads will not be blocked unduly. So this gives us a nice way to scale up our **performance** without using an unrealistic and unreasonable amount of resources to do so, most of which would be wasted.

5.4. Benefits and limitations



Picture 16 Half sync – Half async utilization in ACE framework

Here are some of the **benefits** of this particular pattern:

- **Simplification & performance:** You can simplify some parts of your design quite nicely, it's easy to block in these threads that are spawned. But at the same time, we don't have to have a completely multi-threaded solution that would have too many threads running around without actually improving anything.
- **Separation of concerns:** Synchronization policies in each layer are decoupled so that each layer need not use the same concurrency strategies. This helps to be able to work effectively between those different realms.
- **Centralization of intern-layer communication:** Inter-layer communication is centralized at a single access point, because all interaction is mediated by the queuing layer. We can do all kinds of clever things such as reorder the events, get requests so that they may be able to be run in priority order rather than running in arrival order, to that queue. That could give preference to certain clients that are perhaps higher-paying customers or gold card customers as, as opposed to bronze card customers and so on. So we have a lot of flexibility by the fact that there's this extra queue that's used to mediate between the async part and the sync part that's on top.

Naturally, there are some **limitations** someone has to be aware of:

- **May incur a boundary-crossing penalty:** There can be boundary crossing penalties that are incurred when somebody goes from the proxy, or Async portion, up to the synchronous portion. They'll be additional context switching to do between threads, additional synchronization, data movement costs, if you go between different threads that are running on different cores. There will also typically be dynamic memory allocation and dynamic memory release. So these are things that start to add up if someone is not careful. Of course, if you do long running operations, it's not a big deal, but if you do short running operations, this could be prohibitively expensive from an overhead point of view.

- **Higher-level app services may not benefit from async I/O:** Another downside with this pattern is that the higher level services may not be able to benefit from some efficient asynchronous I/O support that's built into certain operating systems.
- **Complexity and debugging testing:** The problem which we always have whenever we deal with concurrency there's lots of stuff going on. There's multiple threads of control running, and If the async layer is driven by callbacks that itself can be complicated, or interrupts, even more complicated because you have timing properties that are difficult to emulate, debug and test. So when someone use any kind of concurrency pattern, it really pays to think carefully about the different pieces, what the various constraints are, what the various responsibilities are, and use that to help shape the way in which you set up your debugging environment and the way which you think about the various states and interactions in the program.

Chapter 6

Leader/Followers

6.1. Motivation

One of the problems other patterns - such as Half Sync/Half Async - have is that because the thread that receives the incoming request is not the same thread that will process it, we have to end up using dynamic memory allocation, which incurred synchronization overhead and deals with memory fragmentation, free lists and all related issues.

6.2. Problem

Context	Problem
Web servers that run on multi-core & CPU platforms	Half Sync / Half async can incur excessive memory management, synchronization, context switching & data movement overhead in some settings

Table 4 Leader/Followers Context

Likewise, when we try to go ahead and enqueue and dequeue things onto our thread safe synchronized request queue, we're going to have to do synchronization operations. So, we'll have to grab a lock and that'll have some overhead. Also, when we try to move things between these threads, that will typically incur a context switch, which can take many low-level instruction operations in order to be run. We have to deal with caches and other low-level hardware overheads.

- Dynamic memory (de)allocation
- Synchronization operations
- Context switches
- CPU cache updates



Picture 17 Half sync-Half async weaknesses

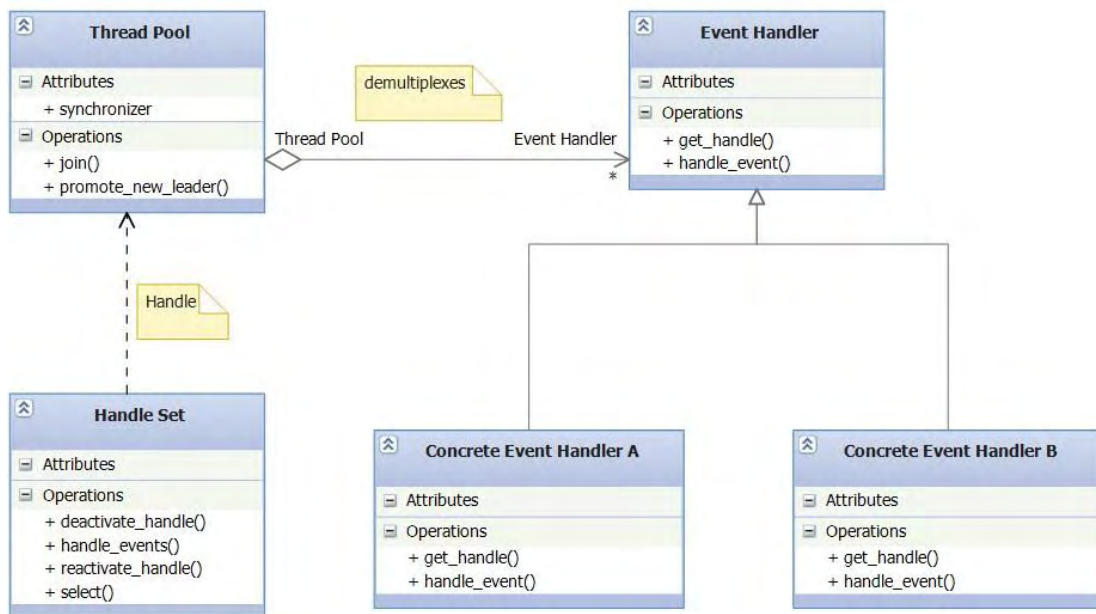
Another issue, speaking of low-level hardware overheads, has to do with the overhead of moving data such as these request messages between caches, between cores, between CPUs, on a multi-CPU platform, and that all starts to add up as well. So even though Half-Sync/Half-Async has many positive qualities for certain types of request, it may be too much overhead.

6.3. Solution

What we're going to describe here is how to apply the Leader/Followers pattern in order to be able to address this issue. Leader/followers is a pattern that allows a **pool of threads**, to efficiently and predictably **take turns** accessing a lower level set of event endpoints.

Leader/Follower pattern provides an efficient and predictable concurrency model where multiple threads take turn sharing event sources to process service requests that occur on them.

This particular pattern is structured in the couple of ways. There is a set of **handles** that you're going to walk and manage, to interact with, and monitor for different kinds and sources of events. Also there is a number of **event handlers** that you can use to dispatch when things happen on those sets of handles. Furthermore a set of concrete event handlers that you can inherit from the **abstract event** handlers to do application or server specific processing logic. We can actually have a **pool of threads**, the handle set in order to take turns, accessing the source of events that handle set encapsulates.



Picture 18 Leader/Followers UML

6.3.1. Dynamics

The dynamics here look a little daunting, but when we break it down, it's actually pretty straightforward. So here is what happens. You end up **spawning a pool of threads** and they all attempt to become the **Leader** thread. Only one thread at a time can become the Leader which will sit there and wait for something to happen on the source of events using **select** and the other threads will then wait on synchronizer or queue as Follower threads.

When an **event occurs**, when an event arrives, the Leader thread will take that request and go ahead and deal with it. It will read it, It will process it and so on, but before it does that, it goes ahead and first promotes one of the **Follower** threads to become a **Leader**.

According to Schmidt [1], in contrast with Leader/Followers, the queuing is typically done in the operating system protocol stack layer. And there is not nearly as much memory available there on a per connection basis. It's usually more like a 100K bytes or so before you end up filling up your Window size.

6.4. Benefits and limitations

Some benefits of Leader/Followers pattern are:

- **Performance enhancements:** We end up getting tremendous performance enhancements in certain areas. For example:
- **Improves CPU cache affinity & removes need for dynamic memory allocation & data buffer sharing between threads:** Having the one thread of control wait for work to do, and then making sure that, that thread is where the work is done can improve cache affinity for the thread, which means that you don't need to move things between the caches. It also means you don't have to even allocate memory dynamically oftentimes if you have a big enough buffer in the stack frame in which the request comes in. So that will improve and reduce certain sources of overhead that Half sync - half async has to incur.
- **Minimizes locking overhead by not exchanging data between threads, thus reducing thread synchronization:** When the data comes, stays with the thread that originally detected it and is going to read it and process it in some subsequent way when it morphs from being a leader thread. So that can reduce the need for synchronization overhead.
- **Does not require a context switch to handle each event:** because we are not moving things around between the threads as much.
- **Programming simplicity:** Simplifies programming of concurrency models where multiple threads receive requests, process responses, & demultiplex connections using a shared handle set.

Some limitations of Leader/Followers pattern are:

- **Implementation complexity:** As Schmint mentions [1] this particular pattern is very complicated to Implement because there's subtleties, there's different variance of Leader/Followers, that are described in the various papers and it's important to understand these different variations to implement it effectively.
- **Lack of flexibility:** It is hard to discard or reorder events because there is no explicit queue
- **Network I/O bottlenecks:** You can't easily discard the messages, you can't easily reprioritize them, because there's no extra queue, there's no extra thread in which to do that. So that can be a little tricky, and of course, because we have one thread of control that's waiting at a time for incoming requests on a set of handles that itself could become a bottleneck in highly scalable systems.

So as a general rule, It is better to use Leader/Followers in situations where we have got more real-time responsiveness, where predictability is more important than scalability and

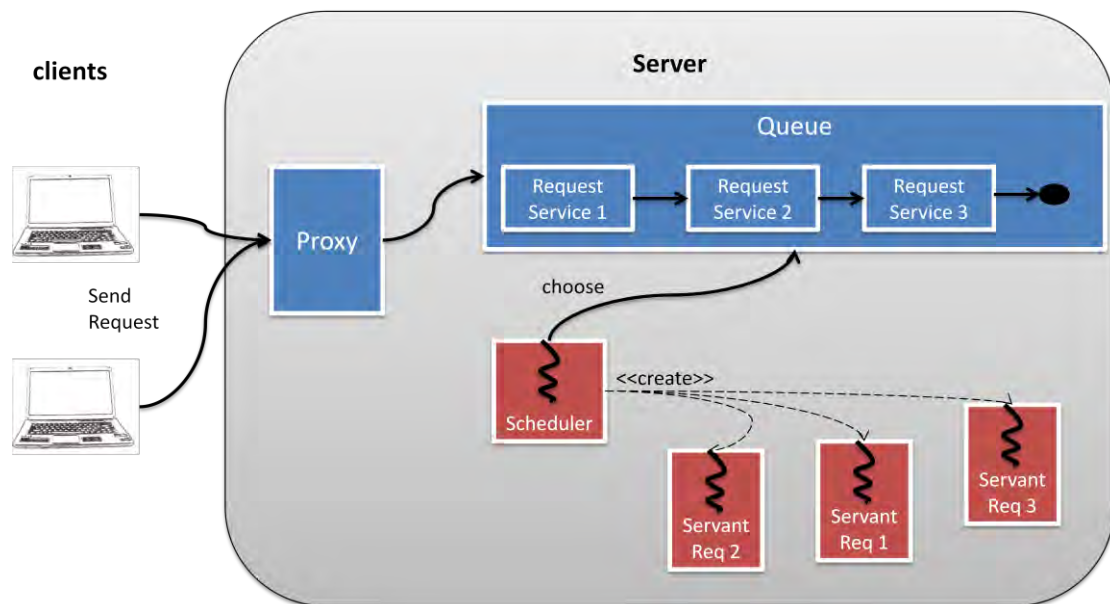
we could use Half-Sync/Half-Async situations with the reverse properties where scalability is very important and predictability is a little bit less important.

Chapter 7

Developing a real server environment using patterns

This chapter is going to describe the development process followed in order to create a real world application using patterns. Also we assess the final features, the restrictions and the software technologies that taking place. The **goal** is to develop a server environment which:

1. will be able to **accept**, **process** and **reply** the client requests effectively
2. provide the **communication** and **synchronization** mechanisms as well as all the important components to be used as a basic layer to further improvements



Picture 20 The Server Environment

7.1. Functionality

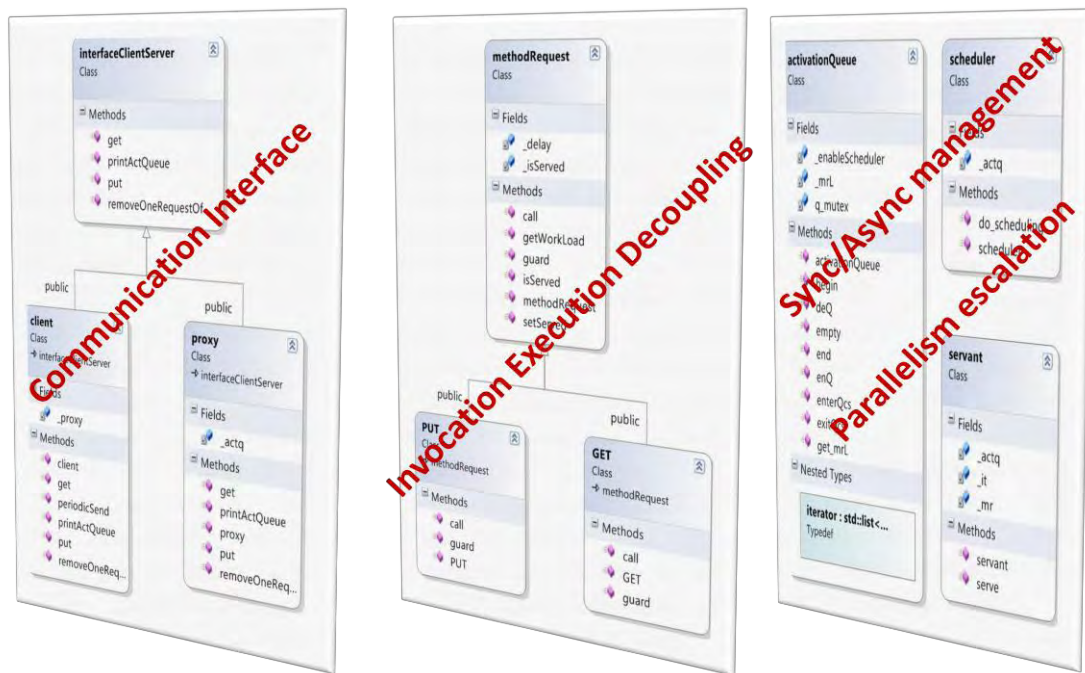
In real world application the challenging task is to decide which of the patterns you are going to use and why. The patterns are the techniques that the application should incorporate, so their use will define the final functionality the applications' architect aims to achieve.

The following are the design features that our server application will have (and that will define the combination of patterns to be used):

- Client service escalation: Every client request has to be served in a separate thread or not if the proxy decides that the workload and the requested type is suitable.
- Synchronous internal service: The internal implementation must provide the mechanisms for sync operations.
- Asynchronous client requests: The acceptance of client requests is asynchronous.
- New service features development facilitation: Basic extensible components able to be developed into new features.

- Concurrency management: Mutual exclusion of critical sections assurance.

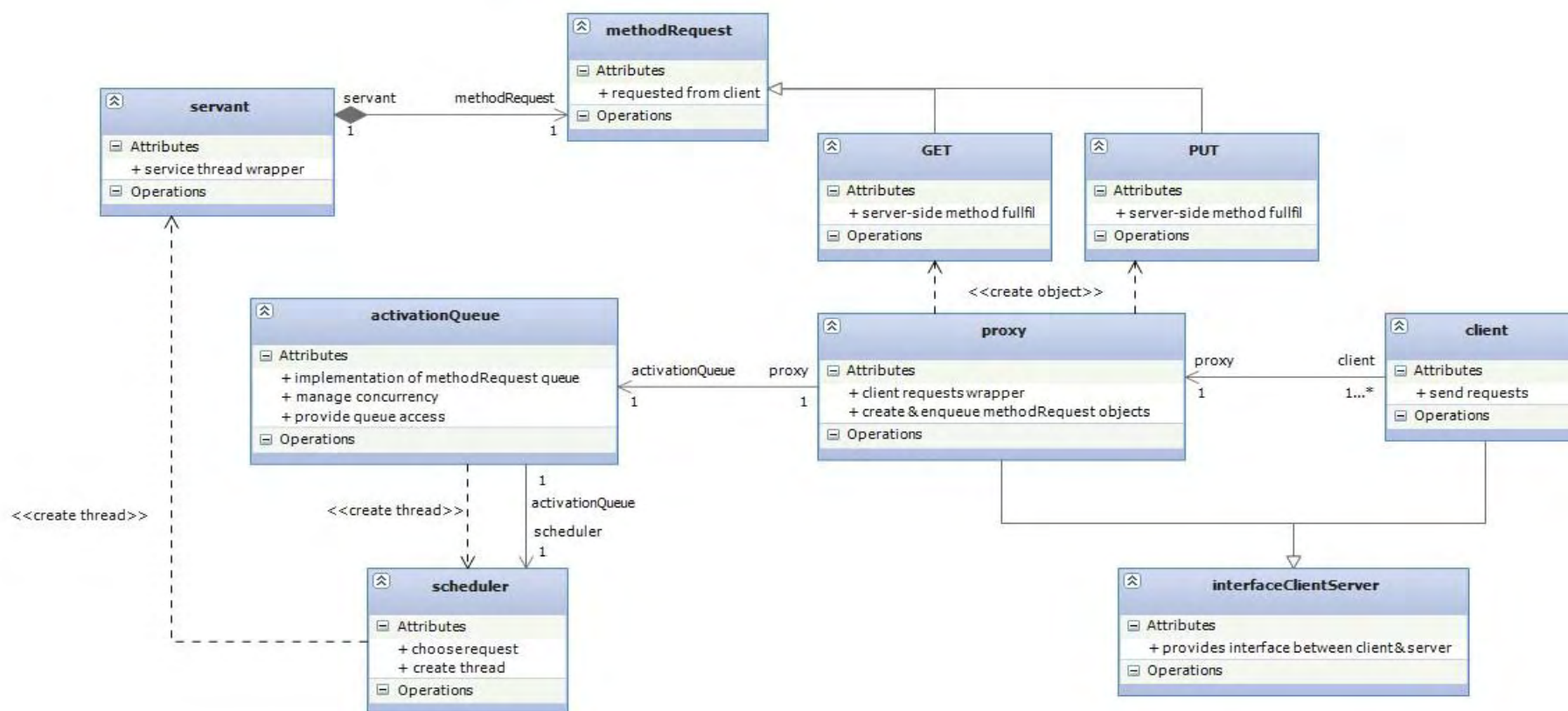
The rest of the chapter describe the way that we achieve this features and build the server.



Picture 21 Server Features

7.2. UML Design diagram

The entire server design diagram that follows, shows the connection type and the relations between the parts. Each parts\ is described in the next section “UML class components diagrams”.



Picture 22 Server Design Diagram

7.3. UML Class components diagrams

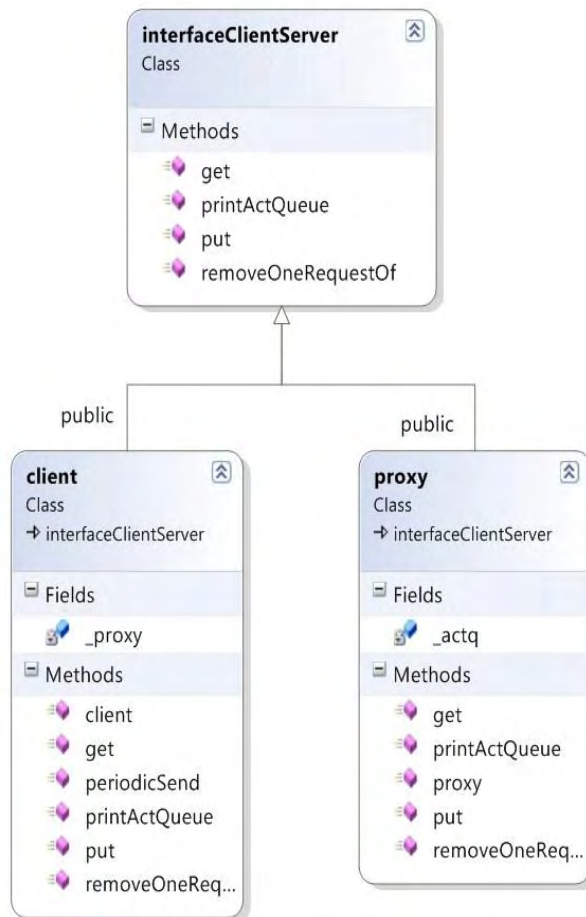
7.3.1. Communication interface

The common communication interface feature is provided from `interfaceClientServer` class. The intention of this class is to provide a common communication interface between the clients and the server so that requested services from the one side have been implemented from the other. This is achieved through pure virtual functions in the base class so that the derived classes will be forced to override and implement these methods.

The next picture depicts the UML class diagram. Here the `client` and `proxy` override the base methods `get()` and `put()` (the others are for debugging purposes) and are forced to use a common communication interface.

See Client p.49

See Proxy p.55



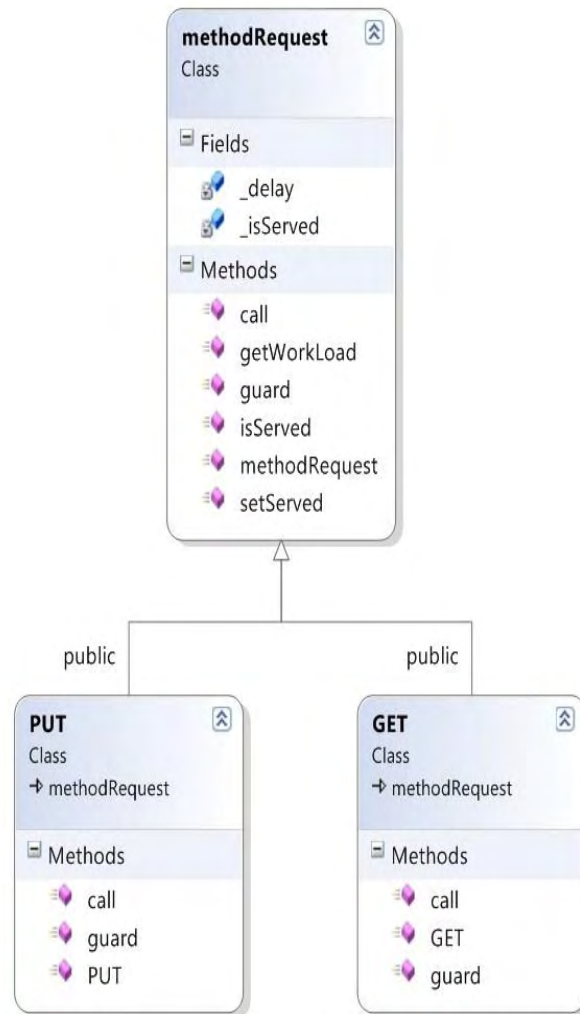
Picture 23 Communication Interface Class Diagram

7.3.2. Invocation – execution method decoupling

Using the feature of decoupling the method invocation from method execution we implemented the `methodRequest` class. This characteristic that was described in Chapter 3 and the Active Object Pattern facilitates the programming and development of the different services from different teams. So, referring to a base class `methodRequest` object, we choose the right service at runtime splitting the development processes of the different server components. The next picture depicts this class management.

See Method Request p.53

See PUT & GET p.59



Picture 24 Method Decoupling Class Diagram

7.3.3. Sync/async management & parallelism escalation

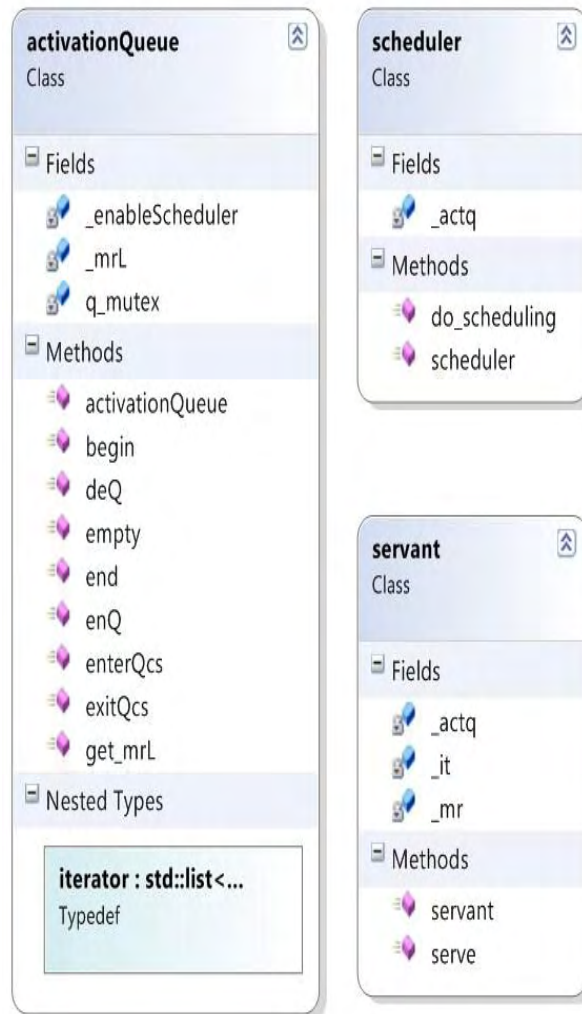
The part of performance escalation and the sync/async management are interrelated. The concurrency that creation of threads imposes and the asynchronous operations resulting from clients requests are managed through the servant and activationQueue components respectively. The queue provides the access methods that are thread safe and synchronize the demanding services. On the other hand the separate thread creation per demanding service is provided by the servant component leading to performance boosting.

The next UML diagram depicts the format of these classes.

See Scheduler p.47

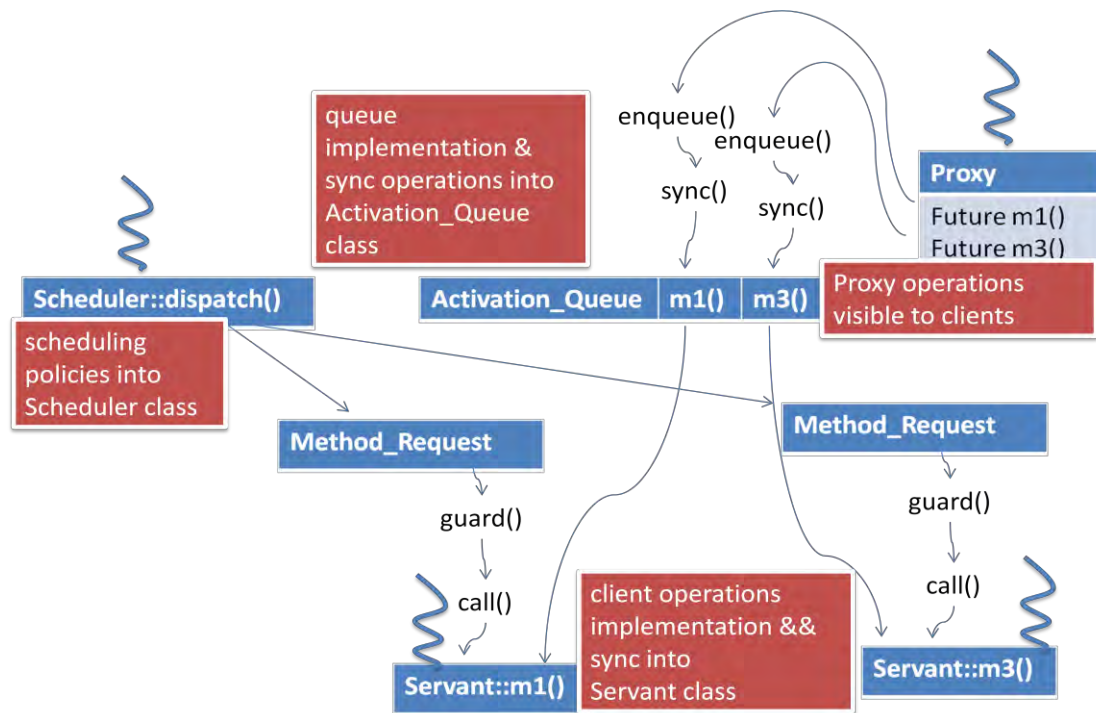
See Servant p.51

See Activation Queue p.57



Picture 25 Sync/Async Management & Parallelism Class Diagram

The following scheme shows an instance of the server in case of servicing two requests m1, m3:



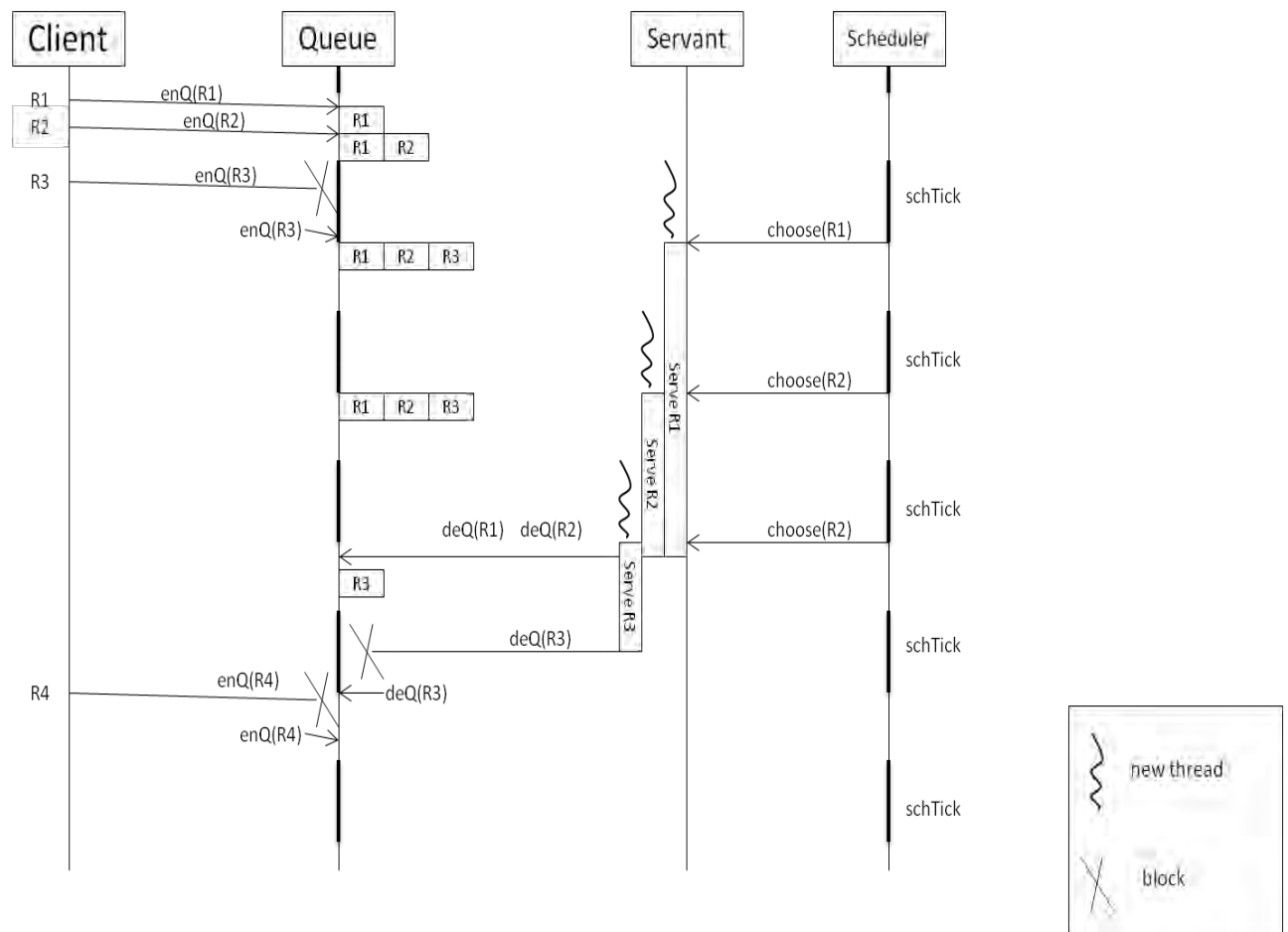
Picture 26 Running Instance

7.4. Concurrency management

The concurrency management is crucial for the stability and the performance of the server. The operations need to be atomic are:

- Every queue management operation like enqueue, dequeue or examining queue
- Every operation between the servants that defined as atomic by the implementation (i.e. from the service developers)

In the following example some concurrency management scenarios are being presented:



Picture 27 Concurrency Management Dynamics

Chapter 8

Conclusion and future work

8.1. About the patterns

I can wholeheartedly conclude that studying patterns makes you a better programmer. It is the content of general accepted solutions and improvements that surely will extend the problem solving strategy someone uses.

Although many patterns are already available, mining new patterns will remain an important activity for the future. Taking into consideration the rapid growth of software engineering industry we can safely assume that in the next decades the software lines required for the systems that are produced will escalate exponentially.

That huge amount of software solutions will demand more reusable and better structured source code. Also, let's not forget that the developing effort (resulting to developing time) is another important factor a programmer must optimize. Using patterns not only could eliminate dramatically this effort but also provide an assurance for the solution correctness.

Moreover, a connection to the web is essential for the upcoming smart phone devices and wireless sensors. For the distributed patterns that means more clients and new areas of research for emerging patterns. Consequently an improvement of the existing solutions and the invention of new ones will be demanded.

8.2. About the application

The application encapsulates some powerful fundamental design features. It facilitates the basic characteristics able to be developed into a dedicated, specific purpose server. Someone or even a team of programmers will be able to implement their scheduling policies, their resource management, their dedicated service fulfillment as simple as building upon the existing software blocs and extending them. The latter is truly important. Giving a connected services layer is the key fact that pattern programming struggles to achieve and the one that the application has managed to provide.

Appendix

Scheduler

```
#ifndef G_SCH
#define G_SCH

/*
 * scheduler.h
 */

namespace server{

    class activationQueue;

    class scheduler{
    public:
        scheduler(activationQueue *aq);
        void do_scheduling();
    private:

        activationQueue *_actq;
    };
}

#endif
```

```

#include <thread>
#include <fstream>
#include <iostream>

#include "scheduler.h"
#include "servant.h"
#include "activationQueue.h"
#include "methodRequest.h"

/*
 * scheduler.cpp
 */
#define SCHED_TICK_SEC 1

namespace server{

scheduler::scheduler(activationQueue *aq){
    _actq = aq;
}

void scheduler::do_scheduling(){

    activationQueue::iterator it; //std::list<methodRequest *>::iterator it;

    std::ofstream myfile;
    myfile.open ("schedulerLog.txt");
    std::cout << "\nFILE scheduler OK\n"<< std::flush;

    std::cout << "\nSTART scheduler\n"<< std::flush;

    while(1){

        _actq->enterQcs();
        if (_actq->empty()){
            myfile << "scheduler found empty activationQ => goto sleep\n";
myfile.flush();
        }
        else{
            for(it = _actq->begin(); it != _actq->end(); ++it){

                if (!((*it)->isServed())){
                    myfile << "method to be served has been choosed\n"; myfile.flush();
                    (*it)->setServed();

                    servant *sr = new servant(_actq, it);
                    std::thread(&servant::serve, sr).detach();

                }
                else{
                    myfile << "scheduler found empty activationQ => goto sleep\n";
myfile.flush();
                }
            }
        }
    }
}
}

```


Client

```
#ifndef G_CLIENT
#define G_CLIENT

#include "interface.h"

//forward declaration
namespace server{
    class proxy;
}

class client: public interfaceClientServer{
public:
    client(void);
    void get(int type);
    void put(int type);
    void printActQueue(void);
    void removeOneRequestOf(int type);

    void periodicSend();
private:
    server::proxy *_proxy; //connection with server proxy
};

#endif
```

```

#include <iostream>

#include "client.h"
#include "proxy.h"

/*
 * client.cpp
 */

namespace server{
    extern proxy *proxyConnection;
}

client::client(void){
    //CS (caution) in case of multiple thread clients init concurrent
    std::cout << "init client " << this << std::endl<< std::flush;
    //initialize server connection per machine/IP
    if (server::proxyConnection == NULL){
        new server::proxy(); //if the first time talking to server
    }
    _proxy = server::proxyConnection; //every client of the same machine connects to
    ONE proxy

    //CS
    return;
}

void client::removeOneRequestOf(int type){
    _proxy->removeOneRequestOf(type);
}

void client::printActQueue(void){
    _proxy->printActQueue();
}

void client::get(int type){
    _proxy->get(type);
}

void client::put(int type){
    _proxy->put(type);
}
}

```

Servant

```
#ifndef G_SRVNT
#define G_SRVNT

#include <list>

/*
 * servant.h
 */

namespace server{
    class methodRequest;
    class activationQueue;

    class servant{
    public:
        servant(activationQueue *actq, std::list<methodRequest *>::iterator it);
        //create a servant for it methodRequest in act activationQueue
        void serve(void);
    private:
        activationQueue *_actq;
        methodRequest *_mr;
        std::list<methodRequest *>::iterator _it; //provide access to methodRequest
        element of activationQueue
    };

}

#endif
```

```

#include <iostream>
#include <fstream>
#include <thread>

#include "servant.h"
#include "methodRequest.h"
#include "activationQueue.h"

/*
 * servant.cpp
 */

namespace server{

    std::ofstream servingFile; //for demonstration serving msgs

    servant::servant(activationQueue *actq, std::list<methodRequest *>::iterator
it){
        _mr = *it;
        _it = it;
        _actq = actq;
    }

    void servant::serve(void){

        int rt;
        //simulate workload delay
        srand (time(NULL));
        rt = rand() % 4 + 1;

        servingFile << "\nserving...\n"; servingFile.flush();

        std::this_thread::sleep_for(std::chrono::seconds(rt));

        _mr->guard();
        _mr->call();
        servingFile << "method workload was... " << rt << "sec\n";
servingFile.flush();
        _actq->deQ(_it);
        delete this;
    }
}

```

Method Request

```
#ifndef G_MTHR
#define G_MTHR

/*
 * methodRequest.h
 */

namespace server{

class methodRequest{
public:
    methodRequest(int delay);

    virtual void guard(void);
    virtual void call(void);

    int getWorkLoad(void);
    bool isServed(void);
    void setServed(void);
private:
    int _delay;
    bool _isServed;
};

}

#endif
```

```
#include <iostream>

#include "methodRequest.h"

/*
 * methodRequest.cpp
 */

namespace server{

    methodRequest::methodRequest(int delay){
        std::cout << "create method request\n"<< std::flush;
        _delay = delay;
        _isServed = false;
    }

    bool methodRequest::isServed(void){
        return _isServed;
    }

    void methodRequest::setServed(void){
        _isServed = true;
    }

    int methodRequest::getWorkLoad(void){
        return _delay;
    }

    void methodRequest::guard(void){
        return;
    }

    void methodRequest::call(void){
        return;
    }

}
```

Proxy

```
#ifndef G_PROXY
#define G_PROXY

#include "interface.h"

/*
 * proxy.h
 */

namespace server{

    class activationQueue;

    class proxy: public ::interfaceClientServer{
    public:
        proxy(void);
        void get(int type);
        void put(int type);

        void printActQueue(void);

        void removeOneRequestOf(int type);

    private:
        activationQueue * _actq;
    };

}

#endif
```

```

#include <iostream>
#include <list>

#include "proxy.h"
#include "activationQueue.h"
#include "PUT.h"
#include "GET.h"

/*
 * proxy.cpp
 */

namespace server{

    proxy *proxyConnection = NULL; //connection interface to clients

    proxy::proxy(void){
        std::cout << "init proxy\n" << std::flush;
        proxyConnection = this; //init client connection
        _actq = new server::activationQueue(); //init activation queue
    }

    void proxy::printActQueue(void){

        std::list<methodRequest *>::iterator it;

        std::cout << "\nprint activation queue\n" << std::flush;

        for(it = _actq->begin(); it!=_actq->end(); ++it){
            std::cout << "mrL element workload: " << (*it)->getWorkLoad() << " isServed:
" << (*it)->isServed() << std::endl << std::flush;

        }
    }

    void proxy::removeOneRequestOf(int type){

        std::list<methodRequest *>::iterator it;

        std::cout << "\nremove request\n" << std::flush;

        for(it = _actq->begin(); it!=_actq->end(); ++it){
            if((*it)->getWorkLoad()==type){
                _actq->deQ(it);
                std::cout << "mrL element workload: " << (*it)->getWorkLoad() << "
removed!" << std::endl << std::flush;
                break;
            }
        }
    }

    void proxy::get(int type){
        std::cout << "\nserver::proxy::get\n" << std::flush;

        int delay = type; //estimate service delay on proxy
    }
}

```


Activation Queue

```

#ifndef G_ACTQ
#define G_ACTQ

#include <list>
#include <thread>

/*
 * activationQueue.h
 */

namespace server{

    class methodRequest;

    class activationQueue{
    public:
        activationQueue();

        //modifiers
        void enQ(methodRequest *mr);
        void deQ(std::list<methodRequest *>::iterator it);

        //iterators
        std::list<methodRequest *>::iterator begin(void);
        std::list<methodRequest *>::iterator end(void);
        typedef std::list<methodRequest *>::iterator iterator;

        //capacity
        bool empty(void);

        //concurrency
        void enterQcs(void);
        void exitQcs(void);

    private:
        std::mutex q_mutex; //provide sync among threads
        bool _enableScheduler;
        std::list<methodRequest *> *_mrL; //shows the list of methodRequest *
    };

}

#endif

```

```

#include <iostream>

#include "activationQueue.h"
#include "scheduler.h"
#include "methodRequest.h"

/*
 * activationQueue.cpp
 */

namespace server{

activationQueue::activationQueue(){
    std::cout << "init activation queue\n"<< std::flush;

    _mrL = new std::list<methodRequest *>; //empty list of methodRequest *

    scheduler *_sched = new scheduler(this); //create a scheduler for this
activationQueue

    std::thread(&scheduler::do_scheduling, _sched).detach();
}

void activationQueue::enQ(methodRequest *mr){
    q_mutex.lock();

    std::cout << "enQ method request\n"<< std::flush;
    _mrL->push_back(mr);

    q_mutex.unlock();
}

void activationQueue::deQ(std::list<methodRequest *>::iterator it){
    q_mutex.lock();
    std::cout << "deQ method request\n"<< std::flush;
    _mrL->erase(it);
    q_mutex.unlock();
}

void activationQueue::enterQcs(){
    q_mutex.lock();
}

void activationQueue::exitQcs(){
    q_mutex.unlock();
}

std::list<methodRequest *>::iterator activationQueue::begin(){
    return _mrL->begin();
}

std::list<methodRequest *>::iterator activationQueue::end(){
    return _mrL->end();
}

bool activationQueue::empty(){
    return _mrL->empty();
}
}

```

PUT & GET

<pre> #ifndef G_PUT #define G_PUT #include "methodRequest.h" /* * put.h */ namespace server{ class PUT: public methodRequest{ public: PUT(int delay); void guard(void); void call(void); }; } #endif </pre>	<pre> #ifndef G_GET #define G_GET #include "methodRequest.h" /* * get.h */ namespace server{ class GET: public methodRequest{ public: GET(int delay); void guard(void); void call(void); }; } #endif </pre>
<pre> #include <fstream> #include <iostream> #include "PUT.h" /* * put.cpp */ namespace server{ extern std::ofstream servingFile; PUT::PUT(int delay):methodRequest(delay){} void PUT::guard(void){ servingFile << "PUT::guard\n"; servingFile.flush(); } void PUT::call(void){ servingFile << "PUT::call\n"; servingFile.flush(); } } </pre>	<pre> #include <fstream> #include <iostream> #include "GET.h" /* * get.cpp */ namespace server{ extern std::ofstream servingFile; GET::GET(int delay):methodRequest(delay){} void GET::guard(void){ servingFile << "GET::guard\n"; servingFile.flush(); } void GET::call(void){ servingFile << "GET::call\n"; servingFile.flush(); } } </pre>

References

- [1] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt-Pattern-Oriented Software Architecture_ A Pattern Language for Distributed Computing (Vol. 4)-Wiley (2007)
- [2] C++ Network Programming, Volume I: Mastering Complexity with ACE and Patterns
- [3] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Mass.: Addison-Wesley, 1995
- [4] GoF Design Patterns - with examples using Java and UML2
- [5] Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2, Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann
- [6] StackOverflow: <http://stackoverflow.com/>
- [7] Boost Libraries <http://www.boost.org/>
- [8] The Adaptive Communication Environment (ACE)
<http://www.cs.wustl.edu/~schmidt/ACE.html>
- [9] Distributed Object Computing (DOC) Group for Distributed Real-time and Embedded (DRE) Systems <http://www.dre.vanderbilt.edu/>
- [10] Wikipedia <http://en.wikipedia.org>