Open Microscopy Environment

Log Service
Software Design Document

May 2003

Andrea Falconi

Swedlow Lab — MSI/WTB Complex
University of Dundee

a.falconi@dundee.ac.uk

Contents

1. Introduction.

This document details the design of the Log Service within the OME Reference Implementation. Relative test cases are described in a separate document.

Before diving into detailed design, let's briefly outline the Log Service required functionality and features. After that, we'll also give an outline of the solution, which is fully described in the next sections.

1.1. Requirements.

The Log Service is required to:

- Allow any class to pass on a log message to be output in a given location (terminal, log file, etc.) by specifying a level of priority for the message. The levels to be supported are, in priority order: DEBUG — all debug messages, INFO — regular log messages that inform about normal application workflow, WARN — messages emitted in case of abnormal or suspect application behavior, ERROR — all error conditions and failures that can be recovered, FATAL — severe failures that require the application to terminate.
- Be flexible and configurable. A configuration file shall provide for fine-tuning of the log settings on a per-class basis. Those settings include the choice of output locations and verbosity based on priority levels.
- Be fast. We want to avoid, as much as possible, to incur into overhead when logging — for example, I/O time and too many context switches. The reason is quite obvious if we think about the high frequency and number of calls to the Log Service from within the application. Even a small per-call overhead might well result in bringing the application to its knees.
- Decouple the application from the actual service implementation. This is important if the implementation heavily relies on somehow instable libraries. The Log Service is virtually called from all the parts of the system and we don't want changes to the implementation (which are likely to be fired by changes in the third-party libraries) to affect the whole system.
- Fit into different concurrency models. The Log Service shall be able to operate the same way either within a single-threaded process or within a multi-threaded process. Moreover, the service shall be able to conveniently synchronize unrelated flows of control in order to maintain data consistency. For example, assuming that the service is configured to output the messages to a log file, a process hosting the UI and another one running from the command line would need to be synchronized somehow, in order to avoid race conditions when writing to the log file.

- Offer an easy-to-use interface to the external classes.

1.2. Solution outline.

The Log Service is implemented as a tiny distributed-objects system, being a *Logger* the only distributed object. This object exports an *ILogger* interface that defines the operations available to any external class for logging and lives in an address space separate from those ones that host the client objects that call upon the service.

The *Logger* itself is an adapter that makes use of the Log4perl [L4P03] library to implement the operations defined by the *ILogger* interface. Log4perl is a Perl port of the popular Log4j [L4J03] library, which has already proven good for several years. Even though Log4perl exhibits most of the useful features of Log4j (it is meant to be an exact clone), at the time of writing, the library is still in alpha release. For this reason, the *ILogger* interface, other than serving the purpose of distribution, will ensure protected variations [Larm01] with respect to changes in the Log4perl library. By building on top of Log4perl, it is trivial to fulfill our goals of being flexible and configurable. Furthermore, Log4perl already supplies the means to pass on a log message to be output in a given location (terminal, log file, etc.) by specifying a level of priority for the message.

For every flow of control — be that a single-threaded process or a thread in a multi-threaded process, there will be a proxy object [GoF95] that is a local representative of the *Logger* object (through the *ILogger* interface) and forwards the calls to the remote *Logger* object. Even though there may be several proxy objects forwarding calls to the *Logger* object at any given point in time, those calls are eventually serialized — within the process hosting the *Logger* object — to ensure synchronization and consistency.

In our specific case, the middleware infrastructure required to support object distribution is pretty simple (with respect to the general case). We directly build it in order to enforce asynchronous invocation semantics and minimize the overhead carried by each call to the Log Service. Upon each call, the proxy objects collect all the required information — such as the log message, information about the caller, etc. — into a log record that is then marshaled in order to be sent into a request to a processor object in the process space hosting the *Logger* object. The processor object unmarshals the request and dispatches the corresponding call to the Logger object. The communication protocol, which also specifies the external data representation, is fairly simple and ASCII encoded. The transport protocol is UDP. This has the advantage that as soon as the operating system puts the data on the local UDP buffer, the proxy is free to carry on and its flow of control doesn't block.

1.3. Document overview.

The following sections in this document will deal with:

- *Object Model*: The core of this document, depicting both the static and dynamic model of the software in terms of objects.
- *Process Model*: In this section, we examine synchronization issues, describe IPC and see how the object model can fit into different concurrency models.
- *Mapping to Code*: How the object model relates to concrete Perl classes and namespaces.
- *Deployment*: Configuration, dependencies, distribution and hardware topology.
- *Failure Model*: Analysis of the possible failures and how such failures are handled.
- *Wrapping up*: We put all the pieces together into a big picture, we explain how to use and configure the Log Service from an outsider's point of view and make some final considerations.

UML [OMG01] diagrams are extensively used throughout this document to precisely depict design. Even though all presented diagrams are commented out and many of them are quite self-explanatory, in order to understand in full the semantics of the diagrams a certain familiarity with UML is necessary. Those that are unfamiliar with UML may want to keep a reference at hand, such as [BRJ00].

2. Object Model.

This section describes both the static and dynamic model of the software in terms of objects. We first introduce the overall logical architecture of the solution model and we show how the solution addresses the requirements. We then dive deeper into detailed object design.

2.1. Overall architecture.

The Log Service is implemented as a tiny distributed-objects system, being a *Logger* the only distributed object. This object exports an *ILogger* interface that defines the operations available to any external class for logging and lives in an address space separate from those ones that host the client objects that call upon the service.
In our specific case, the middleware infrastructure required to support object distribution is pretty simple — with respect to the general case [CDK01]. We directly build it in order to enforce asynchronous invocation semantics and minimize the overhead carried by each call to the Log Service.
The machinery enabling remote method invocations (RMI) is arranged in a client-server fashion and a communication protocol is provided.

Follows a summarized description of the logical structure and behavior of the object model. Focus is on the key elements and on how they relate and cooperate to fulfill the requirements. Notice that what follows is not a detailed description of all elements, relationships and behaviors. This is a bird-eye description that elides many details for the sake of presenting the key ideas to the reader. Detailed static and dynamic models are discussed later.

2.1.1. Structure.

The overall structure of the Log Service is organized according to the Layers pattern [POSA1]. Functionality is sliced at different levels of abstraction (layers), which are stacked according to increasing level of abstraction and are arranged in order to interact with each other according to a strict ordering relation [BBC+00]. Each layer represents a logical partition of the software and provides a cohesive set of functionalities that can be used by others with no concern about the actual implementation.

Our layering scheme is organized as follows (from the top to the bottom):

- *Application*: Contains the object being distributed and its clients. At this level of abstraction, the mechanism used to dispatch a method call to the servant is immaterial to its clients.

- *Distribution*: The machinery to support RMI. By completely hiding the RMI mechanism (total RMI transparency), this layer provides the clients of the *Logger* servant object in the *Application* layer with the illusion of interacting with a local object. Also, it makes immaterial to the servant where its methods are invoked from.
- *Transport*: This layer encapsulates the underlying transport mechanism, UDP, and provides the above layer with a higher level concept of text based communication channel. This is useful as the exchanged messages are ASCII encoded text streams.

The following UML class diagram depicts the overall system structure, the organization into layers and the key elements within each layer:

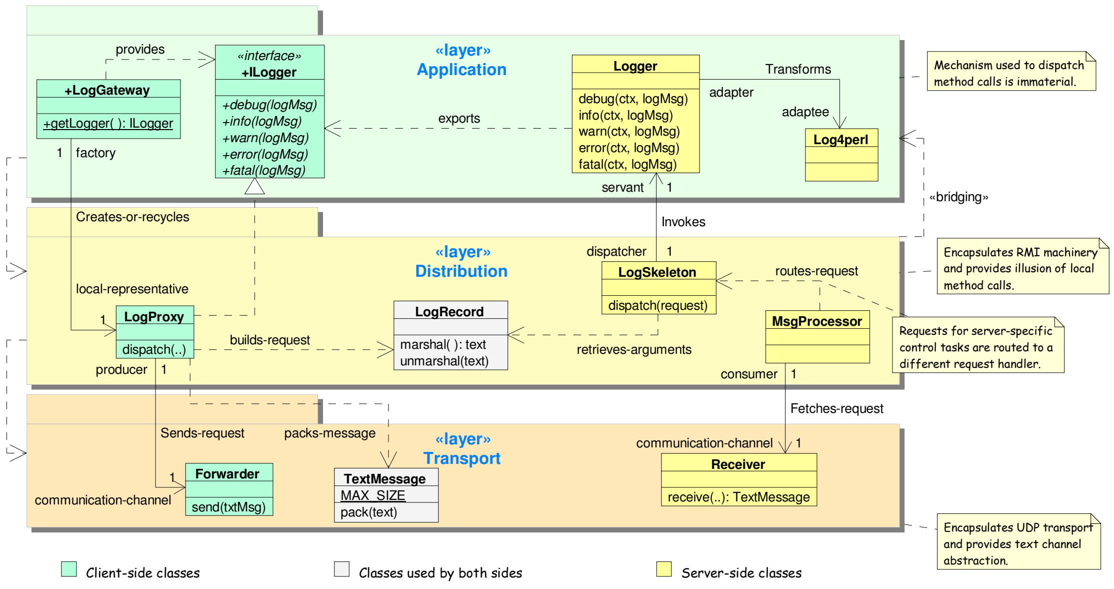


Fig 2-1: Overall static model. Layers are rendered using a stereotyped package. The “layer” stereotype is used to extend the semantics of a UML package to that of a layer and add a new building block to the UML meta-model.

The ordering relation among layers is top-down, meaning that the *Application* layer depends on the *Distribution* layer, which, in turn, depends on the *Transport* layer. Thus, the *Application* layer doesn’t even get to know about the existence of the *Transport* layer.

However, there are a few small exceptions to the dependency chain. Namely, the skeleton and the proxy in the *Distribution* layer will be affected by changes to the servant interface. Controlled and circumscribed dependency that breaks the ordering relation is acceptable and goes under the name of bridging [BBC+00].

Let’s now take a closer look at the key elements within each layer.

The *LogGateway* is the access point for the client classes within the application to the Log Service. It provides client classes with a local representative — the *LogProxy* — of the servant object, the *Logger*. The LogGateway is a factory [GoF95] that initially creates a *LogProxy* object and subsequently recycles this object among client objects that require access to the Log Service.

The *LogProxy* is only known to client classes through the *ILogger* interface, which is exported by the *Logger* to provide access to the Log Service functionalities and is implemented by the proxy [GoF95]. The *LogProxy* is responsible for building an invocation request — represented by the *LogRecord* class — for each call to the *ILogger* implemented operations. The *LogProxy* also packs some information about the log context (such as caller's details and a timestamp) into the invocation request. After building the request, the proxy takes care of marshaling it into an ASCII text stream and passes it on to its communication channel — represented by the *Forwarder* — in order to be sent to the server process hosting the *Logger* object.

The *Forwarder* encapsulates the underlying transport mechanism, UDP, and provides the *LogProxy* with a higher level concept of text based communication channel. This is useful to the proxy as the messages exchanged with the server are ASCII encoded text streams. The exact message formats as well as the communication rules observed by client/server communication are defined by a simple protocol, OME-SLP, which is detailed in the next sections.

The *LogRecord* represents invocation requests made by proxies and is in charge of marshalling/unmarsalling itself into the external data representation defined by the OME-SLP protocol. Marshaled text streams are encapsulated by the *TextMessage* class, that also makes sure they don't exceed the length specified by the *MAX_SIZE* constant.

On the server side, the transport mechanism is encapsulated by the *Receiver*, which pairs up with the *Forwarder* on the client side. As request messages arrive from proxies, they get queued up for retrieval by the *MsgProcessor*. Thus, the *LogProxy* and the *MsgProcessor* are in a producer/consumer relationship by means of the communication channel abstraction provided by the transport layer.

The *MsgProcessor* supervises and coordinates request dispatching. It fetches marshaled requests from the communication channel, obtains the corresponding unmarshaled invocation request objects and passes those objects on to the skeleton in order to dispatch a method invocation to the servant. Requests have to be routed to request handlers (one is the *LogSkeleton* in the diagram) because there are requests other than log requests (represented by *LogRecord*) and request handlers other than *LogSkeleton*. Those requests/handlers are necessary for server-specific control tasks, such as shutdown. This is not explicitly shown in the diagram.

The *LogSkeleton* maps a request to a servant's method, retrieves the log context and message from the request and eventually invokes the servant's method passing those arguments. Thus, the method call requested by the proxy is eventually dispatched to the servant by the *LogSkeleton*, which is, in this respect, the counterpart of the *LogProxy*: they virtually cooperate to dispatch a method call.

The *Logger* is the servant that exports the *ILogger* interface defining the operations available to client classes for logging. It is an adapter [GoF95] that makes use of the Log4perl [L4P03] library to implement the operations defined by the *ILogger* interface. Its methods transform the original call into a suitable call to the Log4perl library. Thanks to the machinery provided by the *Distribution* layer, client objects can transparently invoke the methods of the servant object as if it was a local object. For this reason, we can think of clients and servant virtually exchanging messages.

2.1.2. Dynamics.

The overall behavior of the Log Service during a typical interaction with a client object can be characterized by three phases:

- *Method invocation*: A client object invokes a method defined by the *ILogger* interface, the proxy creates an invocation request, represented by an instance of *LogRecord*, and forwards it to the server-side.
- *Invocation request processing*: The request is received on the server-side and is routed to the skeleton for execution.
- *Method execution*: The skeleton invokes the requested method on the *Logger* servant object.

Notice the separation, both in space (client and server address space) and time, of method invocation from method execution.

The following UML sequence diagram further details a typical interaction:

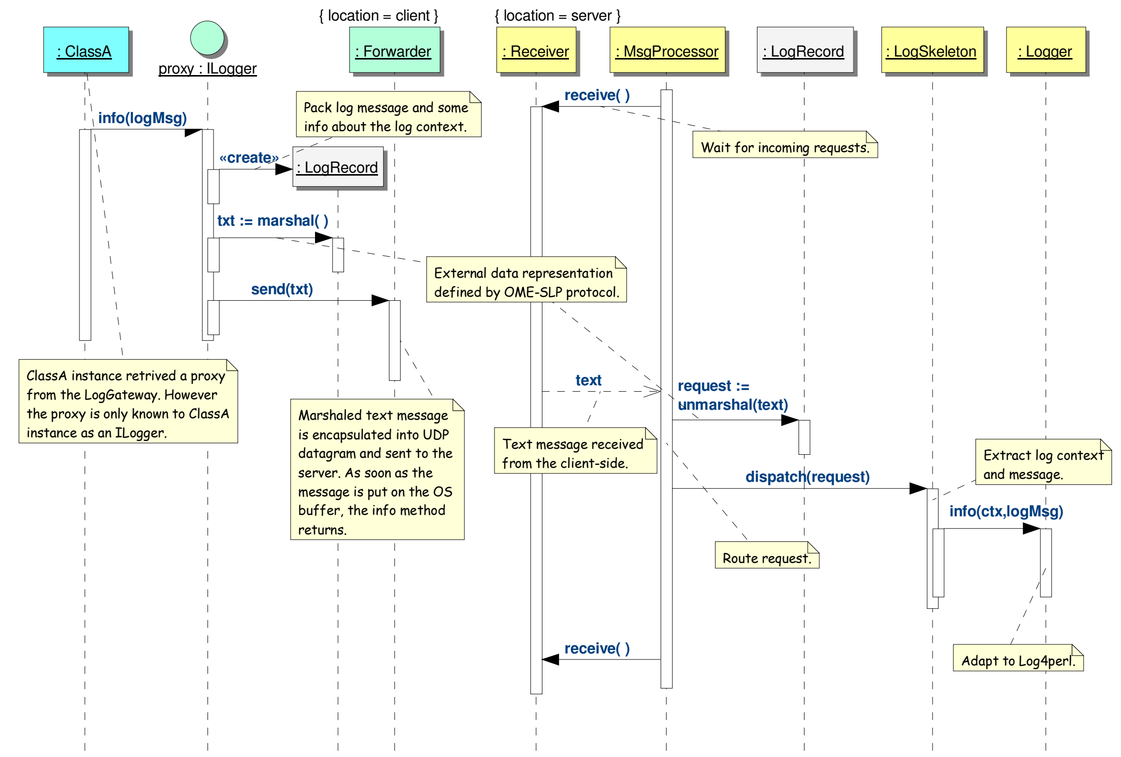

{ location = client }
{ location = server }
: ClassA
proxy : ILogger
: Forwarder
: Receiver
: MsgProcessor
: LogRecord
: LogSkeleton
: Logger
info(logMsg)
Pack log message and some info about the log context.
«create»
: LogRecord
txt := marshal( )
send(txt)
receive( )
Wait for incoming requests.
External data representation defined by OME-SLP protocol.
ClassA instance retrived a proxy from the LogGateway. However the proxy is only known to ClassA instance as an ILogger.
Marshaled text message is encapsulated into UDP datagram and sent to the server. As soon as the message is put on the OS buffer, the info method returns.
text
request := unmarshal(text)
Text message received from the client-side.
dispatch(request)
Extract log context and message.
info(ctx,logMsg)
Route request.
Adapt to Log4perl.
receive( )

Fig 2-2: Overall dynamic model. Notice the half arrow (asynchronous message) used for the info message to denote separation in time from method invocation to method execution. Also notice the tagged values used to denote separation in space (client and server address space).

The diagram depicts a client object (an instance of a given *ClassA*) invoking the *info* method defined by the *ILogger* interface. The client object has previously retrieved an instance of the *LogProxy* (named *proxy* in the diagram) from the *LogGateway* (not shown in the diagram). To the client object’s eyes, the *proxy* is an instance of the *Logger* servant. Upon invocation, the *proxy* creates a new *LogRecord* object to represent the invocation request made by the client object and packs in it the log message specified by the client along with some information about the log context (such as client object's details and a timestamp). The *LogRecord* instance is then asked to provide the ASCII text encoding the marshaled request — the external data representation is defined by the OME-SLP protocol, as already mentioned. At this point, the proxy asks the *Forwarder* object to send this text message to the server-side.
As UDP is the transport mechanism used by the *Forwarder* to send messages, the *info* method returns as soon as the text message is put on the outgoing operating system buffer. The client object relinquishes control straight away, without having to wait for the message to be received on the server-side or for the method to be executed by the servant

(asynchronous RMI).

On the server-side, the *Receiver* object would have already obtained an UDP socket and would have been waiting for incoming UDP datagrams to arrive. The *MsgProcessor* object blocks on the *receive* method, waiting for the *Receiver* to deliver an incoming text message. When UDP datagrams do come in, the *Receiver* queues their content up for retrieval by the *MsgProcessor*. The diagram shows the *MsgProcessor* instance fetching the request sent by the *proxy*. As this request is encoded according to the external data representation dictated by OME-SLP, the *unmarshal* method of the *LogRecord* class is used in order to rebuild a copy of the original *LogRecord* object created by the proxy. At this point the *MsgProcessor* object picks the object that can handle this request, an instance of *LogSkeleton* in this case — we have already mentioned that there are requests/handlers necessary for server-specific control tasks and that requests have to be routed to the right request handler.

Thereafter the *MsgProcessor* object delegates the servant's method execution to the *LogSkeleton* object by invoking the *dispatch* method and by passing the unmarshaled request object. The *LogSkeleton* instance extracts the log context and message from the request in order to invoke the *info* method on the servant. This latter method eventually takes care of adapting the invocation to the Log4perl library, thus fulfilling the service request.

A final consideration on design patterns. The structure and dynamics of the Log Service relates to the distributed variant of the Active Object pattern [POSA2] and to the Forwarder-Receiver pattern [POSA1].

2.1.3. Addressing the requirements.

The reader should have, by now, a grasp of the key ideas within the solution model. Thus, it's a good time to point out how the solution model addresses the Log Service requirements outlined in section 1.1.

The Log Service provides its clients with a fairly simple to use interface — *ILogger*. Clients log messages according to the needed level of priority by invoking the corresponding method defined by *ILogger* — *debug*, *info*, *warn*, *error* or *fatal*. The log message is eventually output in a location specified by a configuration file — terminal, log file, database, and so on.
The above is easily achieved by using Log4perl [L4P03]. Moreover, Log4perl provides the means for fine-tuning of the log settings on a per-class basis.
Also notice that the *ILogger* interface, other than serving the purpose of distribution, will

ensure protected variations [Larm01] with respect to changes in the Log4perl library.

It turns out that the *ILogger* interface and the *Logger* adapter allow the Log Service to:

- Log a message according to its priority and in a given output location.
- Be flexible and configurable.
- Offer an easy-to-use interface to the external classes.
- Decouple the application from the actual service implementation.

Our goal of being fast (on the client side) is achieved by using asynchronous RMI over UDP. Clients of the Log Service relinquish control straight away after invoking one of the *ILogger* methods, without having to wait for the message to be received on the server-side or for the method to be executed by the servant.

The design that we have discussed so far will easily fit into different concurrency models. In fact, the Log Service can operate the same way either within a single-threaded process or within a multi-threaded process. For every flow of control — be that a single-threaded process or a thread in a multi-threaded process, there is a proxy object [GoF95] that forwards calls to the remote *Logger* object by using its own *Forwarder*. No synchronization is thus required on the client side. Even though there may be several proxy objects forwarding calls to the *Logger* object at any given point in time, those calls are eventually serialized by the *MsgProcessor* — within the process hosting the *Logger* object — to ensure synchronization and consistency.

2.1.4. Rationale.

The decision of relying on a third-party library to provide the logging infrastructure is quite obvious — we're not developing a general purpose logging facility, we‘re already too busy with OME. Log4perl [L4P03] seems to offer a lot in terms of functionality and flexibility. Moreover, it is a Perl port of the popular Log4j [L4J03] library, which has already proven good for several years. That's why we preferred it over other candidates.

We don't couple our application directly with Log4perl because, at the time of writing, the library is still in alpha release. This is one reason for wrapping the library with the *ILogger* interface. Notice that, even though the library is in alpha, radical variations are unlikely as Log4perl is a port of Log4j, which is a mature library. Thus, it makes sense to trade-off the big deal of functionality and flexibility that Log4perl provides against possible variations in the library.

There are other reasons for not using Log4perl directly within our code. Namely, this is because we want to:

- Enforce fast asynchronous invocation and minimize the overhead carried by each call to the Log Service. Log4perl provides synchronous invocation, which means, in many cases, that the invoker will have to wait until the log message is output in the desired location. This usually involves overhead due to I/O time.
- Synchronize access to the log resource. In fact, Log4perl, as many excellent Perl libraries, is not thread-safe and doesn't take into account the possibility that different processes may try to access the same log resource, for example a log file, concurrently. This is a problem in our case, as a process hosting the UI and another one running from the command line would need to be synchronized somehow, in order to avoid race conditions when writing to the same log file.
- Exploit processing parallelism.

The above factors can be elegantly resolved by object distribution. In our specific case, the middleware infrastructure required to support object distribution is pretty simple — with respect to the general case [CDK01]. This is one point to build directly the required middleware infrastructure. Reducing to the minimum the overhead carried by each call to the Log Service is another point to it. In fact, our implementation makes use of UDP, which doesn't incur into protocol flow-control (TCP does).
The above trades-off against using a general purpose middleware infrastructure which would require complex configuration (or even hacking) to fulfill the mentioned goals. This would require a few days of work, more or less what is needed to implement our *Distribution* and *Transport* layers.

Another point to building the middleware infrastructure is that of experimenting with Perl threads. In fact, the server-side of the Log Service is multi-threaded. This choice represents a first attempt to explore Perl support to multi-threading and gain useful feedback in order to evaluate the possibility of using Perl threads on a larger scale within the OME Reference Implementation.

2.2. Client side.

Follows a detailed description of the client-side of the Log Service. Classes and relationships are discussed in the static model. The dynamic model addresses the collective behavior of those elements.

2.2.1. Static model.

We describe here the client-side structure, encompassing classes and relationships. The following UML class diagram presents the structure, which is then further detailed, specifying the responsibilities, roles and collaborators of each element.

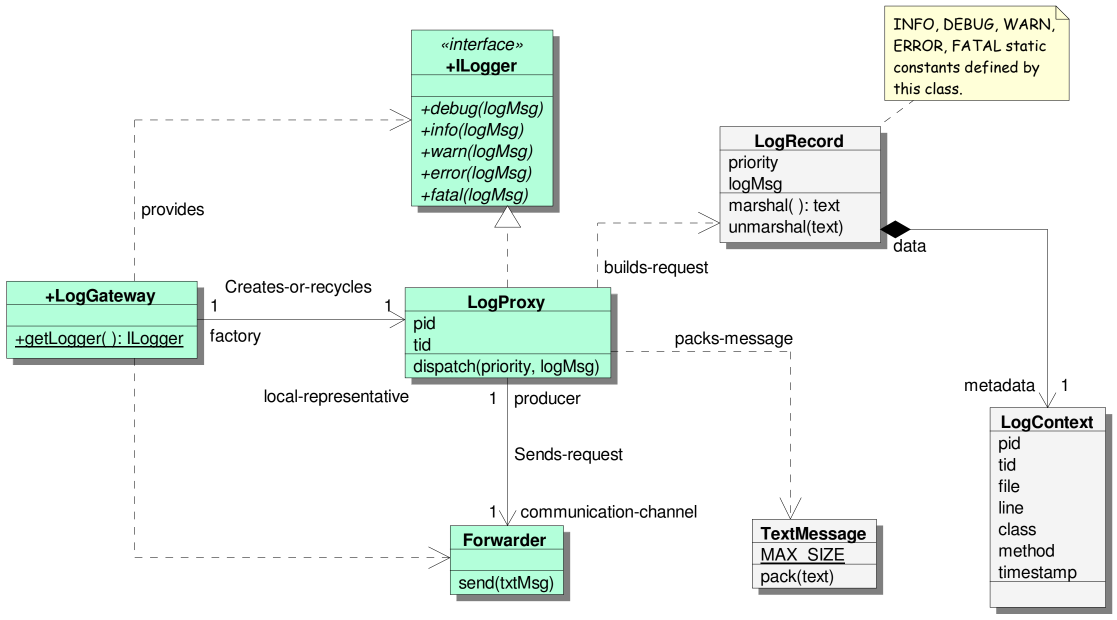


Fig 2-3: Client-side structure. LogGateway and ILogger visibility is explicitly stated as public. These are the only elements known to client classes. The remainder classes have a package visibility that is not explicitly marked.

LogGateway

Access point for the client classes within the application to the Log Service. It provides client classes with a local representative — or proxy [GoF95], the *LogProxy* in our case — of the servant object, the *Logger*. The *LogGateway* is a factory [GoF95] that initially creates a *LogProxy* object and subsequently recycles this object among client objects that require access to the Log Service. The proxy is initialized with a new *Forwarder* instance (this justifies the dependency in the above diagram) and only one *Forwarder* instance is allowed per flow of control — be that a single-threaded process or a thread in a multi-threaded process. That is needed for synchronization purposes (as we‘ll see later) and is enforced by the LogGateway by checking the thread id when clients retrieve the proxy. Client objects retrieve the *Logger* proxy trough the *getLogger* class method.

Responsibilities
- Provide client classes with a proxy to the servant object, the Logger.
- Manage proxy life-cycle.
- Enforce client-side synchronization.

Collaborators
- LogProxy
- Forwarder

Methods
- *getLogger(): ILogger* — Static factory method to retrieve a local representative of the servant. Returned object is a proxy implementing the *ILogger* interface.

ILogger

The interface exported by the *Logger* servant object for distribution. It defines the operations that are available to client objects to access the Log Service.

Methods
- *info*, *debug*, *warn*, *error*, *fatal* — Each of these methods takes a string representing a log message as parameter. Clients log messages according to the needed level of priority by invoking the corresponding method. No return value.

LogProxy

This class implements the *ILogger* interface in order to provide access to the Log Service to client objects. A *LogProxy* instance is a local representative of the *Logger* servant object. The *LogProxy* is responsible for building an invocation request — represented by the *LogRecord* class — for each call to the *ILogger* implemented operations. The *LogProxy* also packs some information about the log context (such as caller's details and a timestamp) into the invocation request by means of the *LogContext* class. After building the request, the proxy marshals it into ASCII text, packs it into a text message (using the *TextMessage* class) and passes it on to its communication channel — represented by the *Forwarder* — in order to be sent to the server process hosting the *Logger* object. The *LogProxy* acts as a message producer to the communication channel.

Responsibilities
- Represent the *Logger* servant to client objects through the *ILogger* interface.
- Build, marshal and send invocation requests on behalf of client objects.

Collaborators
- LogRecord
- LogContext
- TextMessage
- Forwarder

Fields
- *pid* — The process id (integer).
- *tid* — The thread id of the current thread within a process (integer, default to 0 if process single-threaded).

Methods
- *info, debug, warn, error, fatal* — Implementations of the operations defined by the *ILogger* interface. These methods simply invoke the *dispatch* method by passing the relevant priority code, which is stored by the *LogRecord* class, and the log message.
- *dispatch(priority, logMsg)* — Builds, marshals and sends an invocation request. No return value.

LogRecord

The *LogRecord* represents invocation requests made by proxies and is in charge of marshalling/unmarsalling itself into the external data representation defined by the OME-SLP protocol. This is a simple protocol that specifies the exact message formats as well as the communication rules observed by client/server communication and is detailed in the next sections. The invocation request data is stored into the *priority* and *logMsg* fields. Also, some metadata about the caller's context is embedded by means of the *LogContext* class.

Responsibilities
- Represent invocation request, carrying request data and context information.
- Translate objects to external data representation and vice versa.

<u>Collaborators</u>
- LogContext

<u>Fields</u>
- *priority* — The integer code representing the log priority for the invocation request.
- *logMsg* — The log message as a text string.
- *INFO*, *DEBUG*, *WARN*, *ERROR*, *FATAL* — Static constant integers to represent the log priorities.

<u>Methods</u>
- *marshal(): text* — Transforms the current object into an ASCII text string according to the external data representation defined by OME-SLP. This also includes the metadata carried by the *LogContext* instance. Returned value is an ASCII string encoding this *LogRecord* instance.
- *unmarshal(text)* — Reverts a marshaled *LogRecord*, represented by the *text* parameter, back into an object. This also includes the metadata carried by the *LogContext* instance. No return value, this instance represents the unmarshaled object.

LogContext

Convenience class to store some metadata about the context in which a log method is invoked. This includes the caller's details and a timestamp.

<u>Responsibilities</u>
- Hold invocation request metadata.

<u>Fields</u>
- *pid* — The process id (integer).
- *tid* — The thread id of the current thread within a process (integer, default to 0 if process single-threaded).
- *file* — Name of the file containing the caller’s code.
- *line* — Line number within the file containing the caller’s code where the log method was invoked.
- *class* — Fully qualified package (or class) name of the caller.
- *method* — Method or function containing the invocation to the log method.
- *timestamp* — Point in time when the log method was invoked.

TextMessage

Convenience class to store text representing marshaled objects.

<u>Responsibilities</u>

- Hold marshaled objects text and make sure it doesn't exceed *MAX_SIZE*.

<u>Fields</u>

- *MAX_SIZE* — The maximum size of marshaled objects text (integer).

<u>Methods</u>

- *pack(text)* — Stores the passed *text* representing marshaled objects, possibly truncating the text at *MAX_SIZE*. No return value.

Forwarder

This class encapsulates the underlying transport mechanism, UDP, and provides the *LogProxy* with a higher level concept of text based communication channel. This is useful to the proxy as the messages exchanged with the server are ASCII encoded text streams.

<u>Responsibilities</u>

- Encapsulate UDP.
- Send ASCII messages to the server-side.

<u>Collaborators</u>

- UDP API.

<u>Methods</u>

- *send(txtMsg)* — Accepts a *TextMessage* instance as parameter and sends its content to the server-side over UDP. No return value.

2.2.2. Dynamic model.

We now focus on the collective behavior of those elements described in the static model. The client-side classes have to collaborate in order to:

- Provide the application client objects with a *Logger* proxy.
- Forward an invocation request each time a log method is invoked on the proxy.

The first of the above tasks is illustrated by the following interaction scenario:

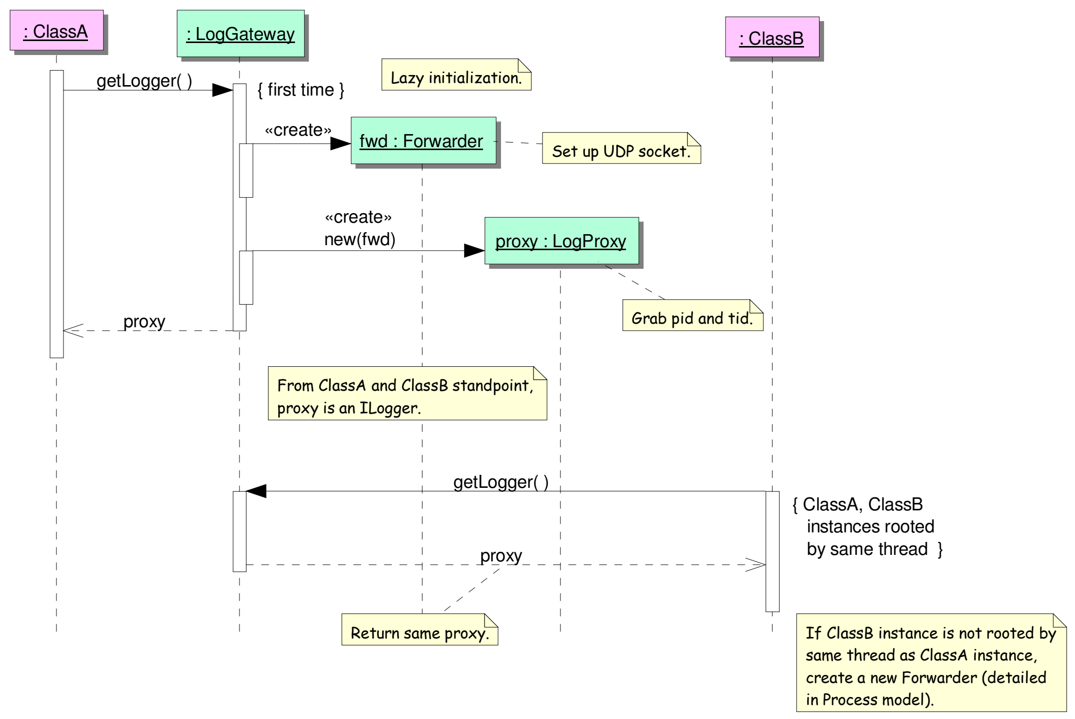

: ClassA
: LogGateway
: ClassB
Lazy initialization.
getLogger( )
{ first time }
«create»
fwd : Forwarder
Set up UDP socket.
«create»
new(fwd)
proxy : LogProxy
Grab pid and tid.
proxy
From ClassA and ClassB standpoint, proxy is an ILogger.
getLogger( )
{ ClassA, ClassB instances rooted by same thread }
proxy
Return same proxy.
If ClassB instance is not rooted by same thread as ClassA instance, create a new Forwarder (detailed in Process model).

Fig 2-4: Retrieving a proxy. Notice the use of constraints to specify that ClassA and ClassB instances are rooted by the same thread and that ClassA instance is the first object to retrieve the proxy within that thread.

The above UML sequence diagram shows an instance of a given client class, *ClassA*, that calls the *getLogger* factory method of *LogGateway*. This is the first time that this method is invoked within the current thread — this is detected by the absence of a link to a *LogProxy* instance. Because of that, the *LogGateway* creates a new *Forwarder* object, *fwd*, which internally sets up the UDP socket needed to connect to the server. Also, a new *LogProxy* is instantiated as *proxy* and linked up with the freshly created *fwd* and cached by the *LogGateway*. The *proxy* is eventually returned to *ClassA* instance that can now use it as a local representative of the servant. At some point later in time, another client object, an instance of *ClassB* rooted by the same thread as *ClassA* instance, calls the *getLogger* method. This time the cached proxy is returned. The same holds for any subsequent call from the same thread.

What happens when *ClassA* instance (or any other client object) invokes a method of the *ILogger* interface on the proxy, is illustrated by the following typical interaction scenario, where *ClassA* instance calls the *info* method.

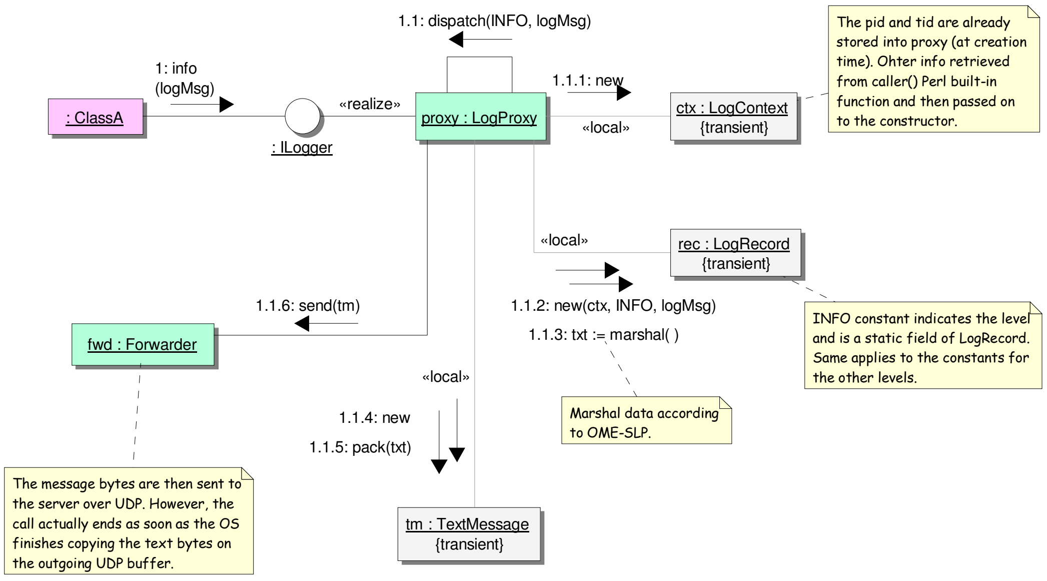

1.1: dispatch(INFO, logMsg)
1: info (logMsg)
: ClassA
: ILogger
«realize»
proxy : LogProxy
1.1.1: new
«local»
ctx : LogContext {transient}
The pid and tid are already stored into proxy (at creation time). Ohter info retrieved from caller() Perl built-in function and then passed on to the constructor.
«local»
rec : LogRecord {transient}
1.1.2: new(ctx, INFO, logMsg)
1.1.3: txt := marshal( )
INFO constant indicates the level and is a static field of LogRecord. Same applies to the constants for the other levels.
Marshal data according to OME-SLP.
1.1.6: send(tm)
fwd : Forwarder
«local»
1.1.4: new
1.1.5: pack(txt)
tm : TextMessage {transient}
The message bytes are then sent to the server over UDP. However, the call actually ends as soon as the OS finishes copying the text bytes on the outgoing UDP buffer.

Fig 2-5: Logging. Notice the use of stereotyped links. The realize stereotype is used to mean the fact that the proxy object exposes the methods defined by the ILogger interface because of the realization relationship between LogProxy and ILogger. The local stereotype marks links that are method-scoped. Also notice the use of the transient tagged value to denote objects that are in existence only for the duration of the interaction.

The *info* method delegates the task of issuing an invocation request to the *dispatch* method of *LogProxy* by passing along the message to be logged and the method id — an integer used to identify the method on the server-side, the *INFO* constant in this case, which is defined by *LogRecord*. The *proxy* collects some information about the caller's context (by using the *caller* built-in Perl function) and stores it into a newly created *LogContext* object, along with the *pid* and *tid*. It next creates a new *LogRecord* to represent the *info* invocation request, links it up with the *LogContext* object and specifies the log message. At this point, the *LogRecord* object is asked to provide the ASCII text representing this invocation request through the *marshal* method. This text is packed into a new *TextMessage* (truncation to *MAX_SIZE* may occur) and passed to the *Forwarder* object, which takes care of sending it over the wire. As soon as the operating system finishes copying the text bytes on the outgoing UDP buffer, the caller relinquishes control.

2.3. Server side.

Follows a detailed description of the server-side of the Log Service. Classes and relationships are discussed in the static model. The dynamic model addresses the collective behavior of those elements.

2.3.1. Static model.

We describe here the server-side structure, encompassing classes and relationships. Responsibilities, roles and collaborators of each element are detailed.

Server-side classes serve two purposes:

- *Invocation request reception and processing*: The server has to wait for incoming invocation requests, has to unmarshal the received ASCII text bytes into a request object and has to route this request to a corresponding request handler for execution. There are two possible types of requests. One is a request to execute a method of the *Logger* servant object. The other is a request to perform a server control task — only server shutdown for the time being, but possibly other tasks in future.
- *Method execution*: The request handler has to invoke the requested method on the servant object, in the case of logging requests, or has to perform server shutdown otherwise.

It is interesting to notice that request processing can be carried out in a manner that is independent from specific request and request handler types. For this reason, we introduce two abstract super-classes: *Request* and *Skeleton*. The first class, obviously enough, represents any request. The *Skeleton* represents any request handler.
The *Request* and *Skeleton* hierarchies are represented in the following UML class diagram:

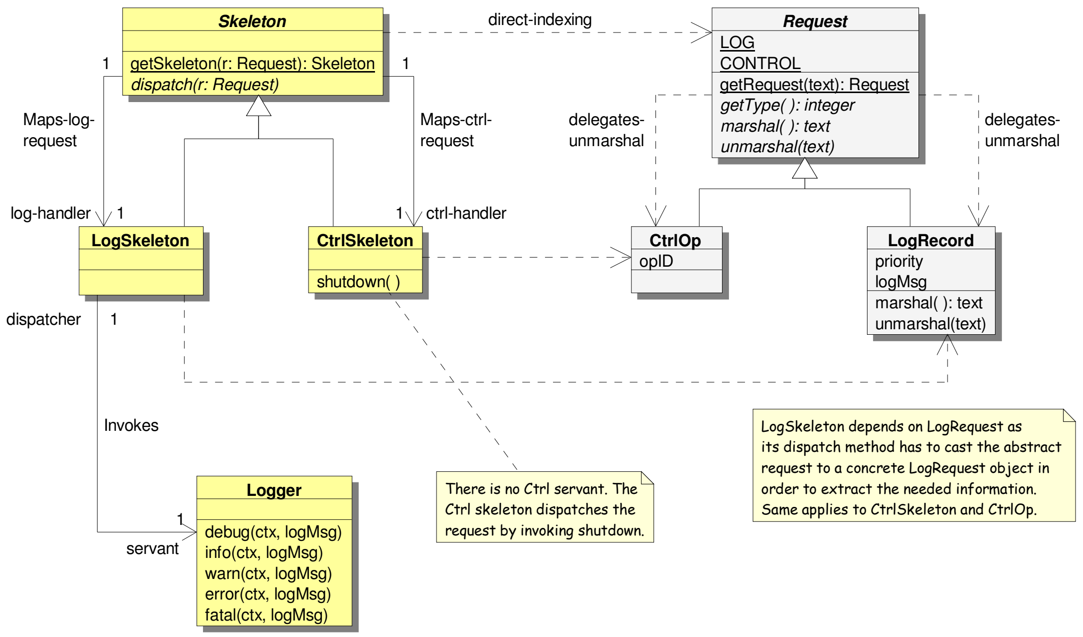


Fig 2-6: Request and Skeleton hierarchies.

The *Request* and *Skeleton* classes decouple the *MsgProcessor* from knowing about concrete requests and request handlers. In fact, as we'll see later, the *MsgProcessor* fetches a marshaled request, asks *Request* to provide an unmarshaled concrete *Request* object (*getRequest* factory method [GoF95]) and asks *Skeleton* to map this concrete request to a request handler (*getSkeleton* method), which is just a *Skeleton* instance to the *MsgProcessor*. The map is built by using direct indexing on an array: the *n*-request handler in the array is the one in charge of handling the request whose id is *n*. This is easily done through the *LOG* and *CONTROL* constants of *Request*.
At this point the *MsgProcessor* asks the concrete *Skeleton* instance to dispatch the concrete request. Notice that the only types known to the *MsgProcessor* are *Request* and *Skeleton*.

The above diagram points out the absence of a servant for server control tasks. In fact, such tasks belong to the *Distribution* layer. Thus, we don't have to provide client classes in the *Application* layer with an interface to control the server-side. Those classes don't have to be aware of the RMI machinery that we're using. The *CtrlSkeleton* will dispatch server control requests (just shutdown for the time being) by invoking its corresponding control method (*shutdown* method).

Fine, but where do server control requests come from? A script will instantiate a *ServerCtrl* class (not shown in the diagram) and will ask it to send a shutdown request.

The *ServerCtrl* class and the other classes in the above diagram are detailed below.

Request

Being the super-class of all requests, *Request* defines the common interface that the *MsgProcessor* and the *Skeleton* use to handle any request. This class has a static factory method [GoF95], *getRequest*, that rebuilds concrete request objects from their external data representation. After retrieving the request type identifier from the external data representation, the task of unmarshaling is delegated to the corresponding *Request* subclass, either *LogRecord* or *CtrlOp*. In this regard, the *getRequest* method can be considered a template method and the *unmarshal* method a hook method, as in the Template Method pattern [GoF95].

Responsibilities
- Represent any request, supplying a common interface.
- Rebuild request objects from external data representation.

Collaborators
- LogRecord.
- CtrlOp.

Fields
- *LOG* — Static constant integer to identify log requests.
- *CONTROL* — Static constant integer to identify server control requests.

Methods
- *getRequest(text): Request* — Static factory method that returns a concrete Request object from the external data representation in *text*.
- *getType()* — Subclasses implement this method in order to return the proper request identifier, either *LOG* or *CONTROL*.
- *marshal(): text* — Abstract method to be implemented by subclasses in order to transform the current object into an ASCII text string according to the external data representation defined by OME-SLP. Returned value is an ASCII string encoding the concrete instance.

- *unmarshal(text)* — Abstract method to be implemented by subclasses in order to revert a marshaled ASCII text, represented by the *text* parameter, back into an object. No return value, the concrete instance represents the unmarshaled object.

LogRecord

This class has already been detailed in the client-side section. The only addictions made here are sub-classing and the *getType* method.

CtrlOp

Subclass of *Request* that represents a control request.

Responsibilities
- Represent control request.
- Translate objects to external data representation and vice versa.

Fields
- *opID* — Set to 0. If new control tasks will be needed, then this field will identify the task.

Methods
Implements the *getType*, *marshal* and *unmarshal* methods as explained in *Request*.

ServerCtrl

Convenience class to control the server process hosting the servant. Only server shutdown is implemented for the time being, but possibly other control tasks may be needed in future. Upon creation, a new *Forwarder* is configured in order to be able to send control requests afterward. When the *shutdown* method is invoked, a new *CtrlOp* object is created and then its marshaled representation is wrapped into a *TextMessage* and sent over the wire to the server by means of the *Forwarder*.

Responsibilities
- Send shutdown request.

Collaborators
- CtrlOp.

- TextMessage.
- Forwarder.

<u>Methods</u>
- *shutdown()* — Sends the shutdown request as outlined above. No return value.

Skeleton

Being the super-class of all request handlers, *Skeleton* defines the common interface that the *MsgProcessor* uses to dispatch any request to the corresponding handler. This class has a static method, *getSkeleton*, that maps concrete request objects to the corresponding request handler, either *LogSkeleton* or *CtrlSkeleton*.

<u>Responsibilities</u>
- Represent any request handler, supplying a common interface.
- Map concrete request objects to corresponding handler.

<u>Collaborators</u>
- LogSkeleton.
- CtrlSkeleton.

<u>Methods</u>
- *getSkeleton(Request): Skeleton* — Static method that, given a concrete request object, returns the corresponding request handler.
- *dispatch(Request)* — Abstract method to be implemented by subclasses in order to invoke the method specified by the concrete *Request* passed in. No return value.

LogSkeleton

This class implements the *Skeleton* abstract interface to handle log requests. The *LogSkeleton* maps a log request to a *Logger* servant's method, retrieves the log context and message from the request and eventually invokes the servant's method passing those arguments. Thus, the method call requested by the proxy is eventually dispatched to the servant by the *LogSkeleton*, which is, in this respect, the counterpart of the *LogProxy*: they virtually cooperate to dispatch a method call.

<u>Methods</u>
- *dispatch(Request)* — Casts the input parameter to a *LogRecord* and then handles the request as explained above. No return value.

CtrlSkeleton

This class implements the *Skeleton* abstract interface to handle server control requests. The *CtrlSkeleton* maps a control request to its *shutdown* method, which is invoked to fulfill the request.

Methods
- *dispatch(Request)* — Casts the input parameter to a *CtrlOp* and then handles the request as explained above. No return value.
- *shutdown()* — Performs server shutdown. No return value.

Logger

The *Logger* is the servant that exports the *ILogger* interface defining the operations available to client classes for logging. It is an adapter [GoF95] that makes use of the Log4perl [L4P03] library to implement the operations defined by the *ILogger* interface. Its methods transform the original call into a suitable call to the Log4perl library. Thanks to the machinery provided by the *Distribution* layer, client objects can transparently invoke the methods of the servant object as if it was a local object. For this reason, we can think of clients and servant virtually exchanging messages.

Methods
- *info, debug, warn, error, fatal (ctx, logMsg)* — Merge the input parameters into a string and pass it to the corresponding method of Log4perl. No return value.

The classes in the *Request* and *Skeleton* hierarchies as well as the *Logger* are involved in the method execution task mentioned before. The reception and processing task is undertaken by the *Receiver* and *MsgProcessor*. The *Receiver* is responsible for collecting messages from the network and buffering them — with the help of the *MsgQueue* class — for later fetching by the *MsgProcessor*, which supervises and coordinates request dispatching. In doing so, it delegates tasks to the *Request* and *Skeleton* classes.

The following UML class diagram illustrates and further details the above.

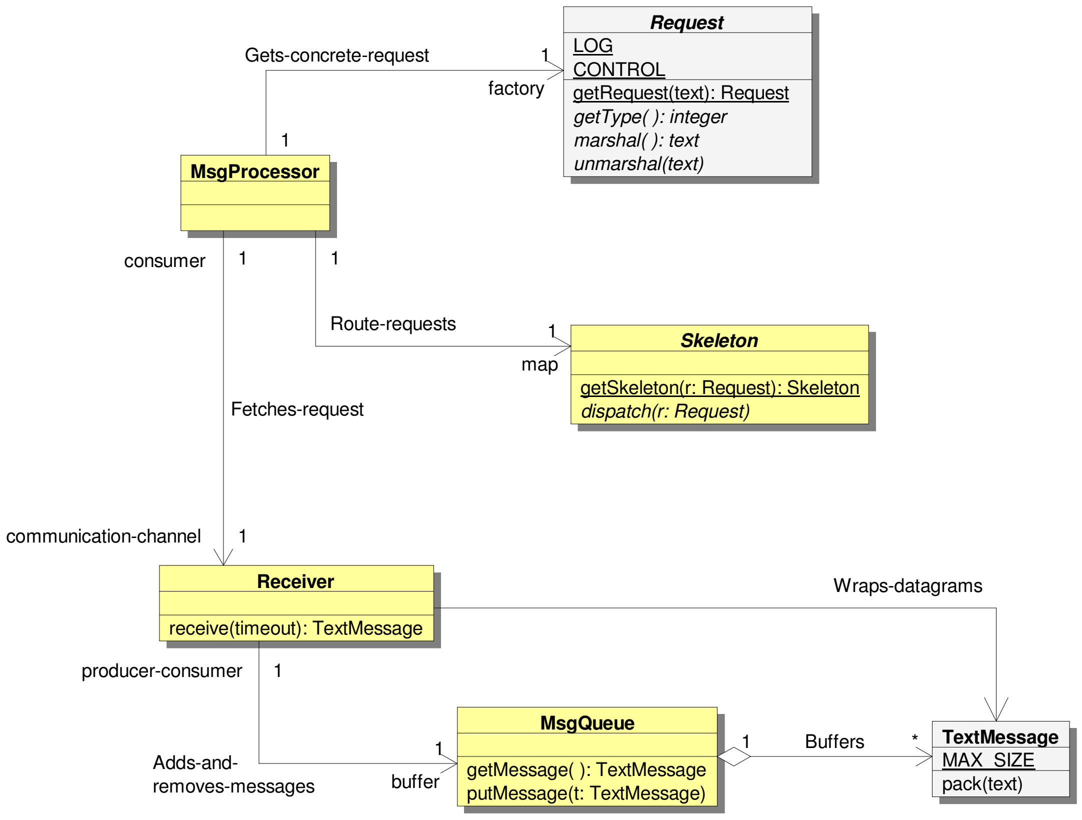


Fig 2-7: Structure of the participants in the reception and processing task.

Let's take a closer look at the *MsgQueue*, *Receiver* and *MsgProcessor* classes.

MsgQueue

This is a synchronized unbounded queue that buffers *TextMessage* instances. Only one tread at a time is allowed to access the queue as in the Monitor Object pattern [POSA2]. Thus, the *getMessage* and *putMessage* method execution is synchronized to ensure that only one method at a time runs within the *MsgQueue* object.

<u>Responsibilities</u>

- Buffer *TextMessage* instances.
- Synchronize access to the queue.

<u>Collaborators</u>

- TextMessage.

<u>Methods</u>

- *getMessage(): TextMessage* — Returns the message at the head of the queue. If the queue is empty, then a null value is returned.
- *putMessage(TextMessage)* — Adds the input message to the queue (at the tail). No return value.

Receiver

This class encapsulates the underlying transport mechanism, UDP, and provides the *MsgProcessor* with a higher level concept of text based communication channel. As UDP datagrams come in, their content is wrapped into a *TextMessage* object which is then added to the message queue (managed by *MsgQueue*) for retrieval by the *MsgProcessor*. The *Receiver* regards the *MsgQueue* as a buffer and it acts both as a producer and as a consumer. It is a producer because it adds messages to the queue and a consumer because it removes messages from the queue on behalf of the *MsgProcessor* (*receive* method).

<u>Responsibilities</u>

- Encapsulate UDP.
- Receive and queue messages.
- Removes messages from the queue on behalf of the *MsgProcessor*.

<u>Collaborators</u>

- MsgQueue.
- TextMessage.
- UDP API.

<u>Methods</u>

- *receive(timeout): TextMessage* — Fetches a message from the queue by calling the *getMessage* method of *MsgQueue*. If the queue is empty, the caller is blocked until a message is added to the queue or *timeout* elapses. If *timeout* elapses and the queue is still empty, then a null value is returned.

MsgProcessor

The *MsgProcessor* supervises and coordinates request dispatching. It fetches marshaled requests from the communication channel — represented by the *Receiver*, obtains the corresponding unmarshaled request objects from *Request* — which acts as a factory [GoF95] by providing concrete requests objects, and passes those objects on to *Skeleton* in order to route those requests to the corresponding request handler. The *Skeleton* maps *LogRecord* objects to the *LogSkeleton* instance and *CtrlOp* objects to the *CtrlSkeleton* instance. After obtaining the right request handler, the *MsgProcessor* can dispatch the request.

<u>Responsibilities</u>
- Supervise and coordinate request dispatching.

<u>Collaborators</u>
- Receiver.
- Request.
- Skeleton.

2.3.2. Dynamic model.

We now focus on the collective behavior of those elements described in the static model. The server-side classes have to collaborate in order to:

- *Initialize the server process*: Processing resources have to be acquired, objects have to be created, initialized and linked according to the structure described in the static model.
- *Receive and process requests*: As UDP datagrams come in, their content has to be buffered and then unmarshaled into a concrete request object, which has to be routed to the corresponding request handler for request fulfillment.
- *Fulfill requests*: The request handler has to invoke the requested method on the servant object, in the case of logging requests, or has to perform server shutdown otherwise.
- *Terminate the server process*: Acquired resources have to be released, the receiving and processing activities have to terminate.

Let's analyze each of the above collaborations in detail.

Initialization

The initialization code will allocate two flows of control, one for the *Receiver* object and the other for the *MsgProcessor* object, will then create and initialize those objects, and will finally start their processing loops. This initialization code is contained in a convenience class, *LogService*. The *Receiver* initialization code will set up the message queue, the UDP socket and will start listening for incoming requests. The *MsgProcessor* object is passed a reference to the *Receiver* object and will access the static methods of the *Request* and *Skeleton* classes, no instance of those classes is therefore needed and the links to those classes are implicit because of global visibility. However, the *Skeleton* class requires some static initialization in order to create and link the *LogSkeleton* and *CtrlSkeleton* objects. A static initialization method of the *Skeleton* class will do the job when invoked by the *MsgProcessor* initialization code. The *CtrlSkeleton* object needs to be linked to the *Receiver* and *MsgProcessor* objects in order to perform server shutdown. Thus, the *MsgProcessor* object will pass a reference to itself and to the *Receiver* object to the *Skeleton* static initialization method. Upon creation, the *LogSkeleton* object will create, link and initialize the *Logger* servant object, which, in turn, will configure Log4perl.

Reception and processing

The *Receiver* object runs in its own flow of control and blocks on its UDP socket waiting for incoming datagrams. When datagrams do come in, their content is wrapped into a new *TextMessage* object which is then added to the message queue by invoking the *putMessage* method. The *Receiver* object carries on in this processing loop until the server is shut down.
In the mean time, the *MsgProcessor* object, which also runs in its own flow of control, is trying to fetch any pending message from the queue — by invoking the *receive* method of *Receiver*. The *receive* method regularly polls the queue by calling the *getMessage* method. When a message is available, this is returned to the *MsgProcessor* object. If no message comes in within a timeout specified by the *MsgProcessor* object, then the *receive* method gives up and returns a null value. This timeout is needed in order to allow the *MsgProcessor* to check when it's time to exit its processing loop. That is detailed in the Process Model.

Here is a typical interaction scenario:

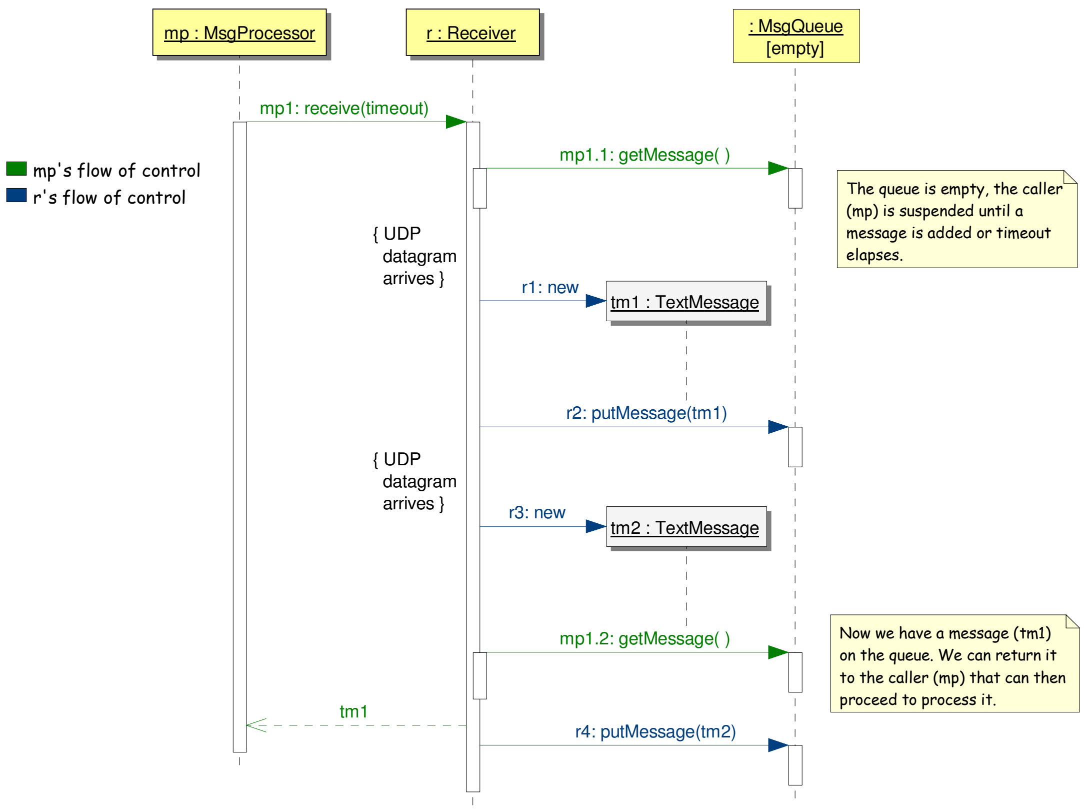


Fig 2-8: Message reception. MsgProcessor and Receiver objects are active objects. The name of the flow of control is prefixed to each message number in order to distinguish one flow from another (also, different colors are used).

The above UML sequence diagram gives an idea of how the *Receiver* and *MsgProcessor* objects run concurrently and coordinate their activity. Also, the *Receiver* processing loop is clearly outlined as well as the work carried out by the *receive* method, which runs in the *MsgProcessor* object's flow of control. All these topics will be further discussed in the Process Model.

What happens after the *MsgProcessor* object has obtained a message from the *Receiver* object? The message is unmarshaled into a concrete request object and the request is routed and dispatched to the corresponding request handler. Thus, the *MsgProcessor* processing loop consists in fetching a marshaled request from the *Receiver*, unmarshaling, routing and dispatching the request. The *MsgProcessor* object carries on in this processing loop until the server is shut down.

The following UML collaboration diagram shows what happens when a log request is received:

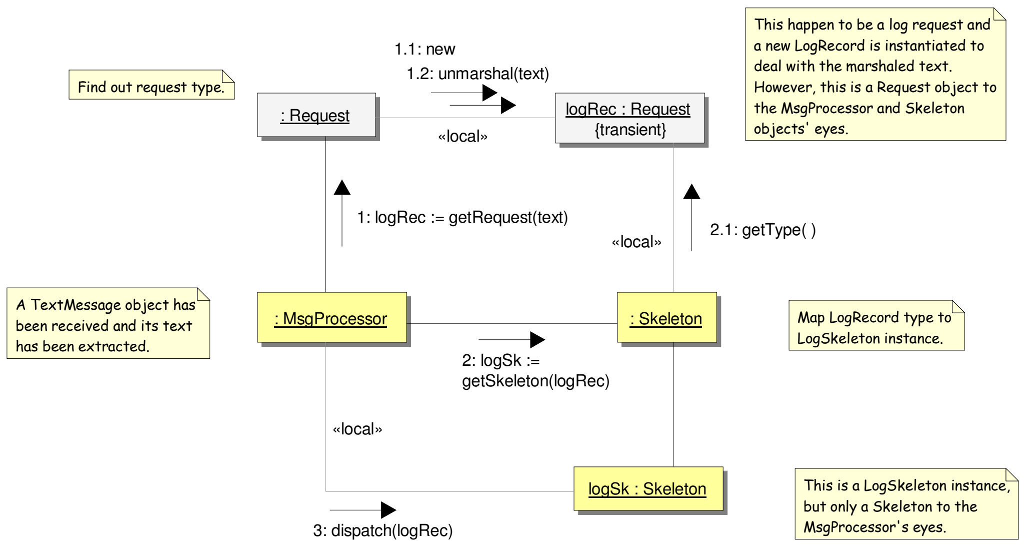

Find out request type.
: Request
1.1: new
1.2: unmarshal(text)
«local»
logRec : Request
{transient}
This happen to be a log request and a new LogRecord is instantiated to deal with the marshaled text. However, this is a Request object to the MsgProcessor and Skeleton objects' eyes.
1: logRec := getRequest(text)
«local»
2.1: getType( )
A TextMessage object has been received and its text has been extracted.
: MsgProcessor
: Skeleton
Map LogRecord type to LogSkeleton instance.
2: logSk := getSkeleton(logRec)
«local»
logSk : Skeleton
This is a LogSkeleton instance, but only a Skeleton to the MsgProcessor's eyes.
3: dispatch(logRec)

Fig 2-9: Unmarshaling, routing and dispatching a request. The local stereotype marks links that are method-scoped. The transient tagged value states that the logRec object is in existence only for the duration of the interaction.

After fetching the *TextMessage* object from the *Receiver* object, the *MsgProcessor* object extracts the text string encoding the marshaled request and asks *Request* to supply the corresponding unmarshaled request objects (*getRequest* method). *Request* finds out the request type, creates a concrete request object of that type (this happens to be *LogRecord* in the diagram) and then delegates unmarshaling. Having obtained a concrete request (which is still a *Request* to the *MsgProcessor*), the *MsgProcessor* object passes it on to *Skeleton* in order to route the request object to the corresponding request handler (*getSkeleton* method). The *Skeleton* maps *LogRecord* objects to the *LogSkeleton* instance and *CtrlOp* objects to the *CtrlSkeleton* instance. Thus, a reference to the *LogSkeleton* instance is returned to the *MsgProcessor* in the above collaboration. After obtaining the right request handler, the *MsgProcessor* can dispatch the request.

Request fulfillment

After a request is dispatched to the corresponding request handler, it is fulfilled as already explained in the *LogSkeleton*, *Logger* and *CtrlSkeleton* class specifications.

Termination

Upon dispatching of a shutdown request, the *CtrlSkeleton* will invoke its *shutdown* method, which will ask the *Receiver* and *MsgProcessor* objects to terminate execution. The *Receiver* and *MsgProcessor* both expose an *exit* method that causes the object to leave its processing loop, stop its activity and release its flow of control. This is detailed in the Process Model. The *Receiver* also releases the UDP socket.

A final note. Some trivial issues (such as routine initialization code, constructors and destructors, accessors and mutators or, in general, features that are required to write working source code, but that can be trivially derived from this object model) haven‘t been addressed by this object model explicitly. Those are routine programming tasks that would add little to the semantics of this model — but would inflate this document quite a bit, and for this reason are omitted.

3. Process Model.

In this section, we examine synchronization issues, describe IPC and see how the object model can fit into different concurrency models.

3.1. Flows of control and synchronization.

We have already mentioned some concurrency issues in the object model. Now it's time to completely unfold that matter. Before talking about processes and threads, it's worth studying concurrency from a logical and general point of view. Specific process and thread semantics introduce many details that would add nothing to the concurrency issues that we're going to study, but would simply obfuscate the matter. We discuss later allocation of flows of control to processes and threads as well as specific issues relative to the Perl environment.

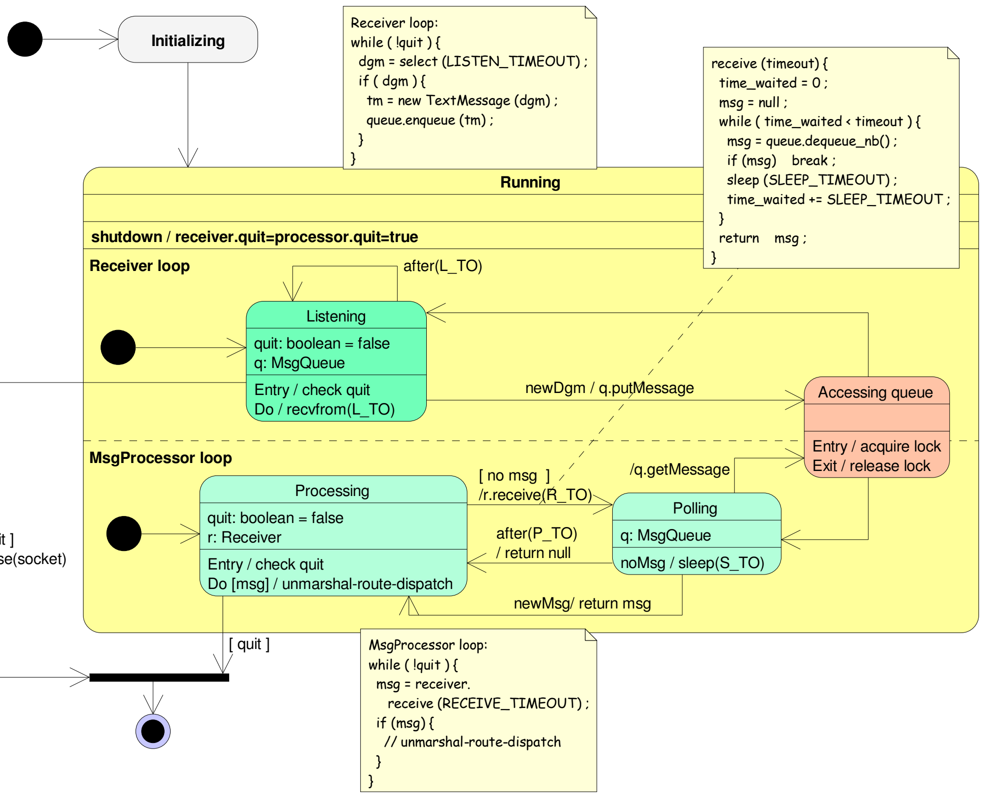
As far as the client-side goes, we don't want to make assumptions on the number of threads running within a process. Thus, the client-side objects could be rooted by just one flow of control — in the case of a single-threaded process, or different flows of control might be sharing some objects — possible in a multi-threaded process. The only objects that would cause consistency problems if accessed concurrently are the *Forwarder* and the *LogGateway*. The first one would cause problems because different flows of control might be overlapping while writing to the UDP socket managed by the *Forwarder*. For a similar reason, the code that provides lazy initialization in the *LogGateway* needs to be synchronized.

No matter how many threads are running within the client process, we only allow one *Forwarder* per flow of control. As a result, every flow of control (be that a single-threaded process or a thread in a multi-threaded process) — will be able to write to its own UDP socket without having to synchronize with other concurrent flows of control, if any. Moreover, we don't incur into synchronization overhead, in fact different flows of control don't have to compete to access a shared synchronized *Forwarder*.
The *LogGateway* could be organized according to the Thread-Specific Storage pattern [POSA2], in order to allow different flows of control to use one global access point to retrieve an object that is local to a thread, without incurring synchronization overhead. However, Perl threads automatically enforce a mechanism that turns out to have a similar effect. We'll detail that later along with how to enforce only one *Forwarder* object per-thread when Perl threads are used. Obviously enough, in a single-threaded process the only *Forwarder* instance can not possibly be shared and access to the *LogGateway* doesn't need to be synchronized.

Even though all processes that host the client-side objects could be single-threaded, we'll definitely have to take into account the possibility of having several of those processes active at the same time — for example, think about the Web UI run by *mod_perl* and about accessing OME through the CLI. Thus, in general, we have to allow for different flows of control on the client-side that concurrently send logging requests to the server. This is not a problem if those request are eventually serialized in order to access to Log4perl serially. The server takes care of that.

Server-side concurrency is a bit more complicated due to the fact that, in order to improve performance, we have two concurrent flows of control that need share some data. Specifically, one flow is rooted by the *Receiver* and the other one is rooted by the *MsgProcessor*. The *Receiver* cyclically waits for incoming UDP datagrams and buffers their content on the *MsgQueue*. The *MsgProcessor* cyclically fetches pending messages from the queue (by invoking the *receive* method of *Receiver*), unmarshals them into request objects, routes and dispatches those requests to the corresponding request handler. So, it turns out that access to the *MsgQueue* object need to be serialized in order to maintain consistency. This can be easily achieved by means of a lock. Before a flow of control can enter the queue, it has to acquire a lock. This lock may be held by just one flow of control at a time. When the operation on the queue completes, the lock is released and the other flow can acquire it. To implement the queue, we'll use a Perl thread-safe queue that exploits a behavior similar to that in the Monitor Object pattern [POSA2]. This is detailed later.

The server behavior is modeled with a concurrent finite state machine, that we present in the following UML statechart:



```
Receiver loop:
while ( !quit ) {
  dgm = select (LISTEN_TIMEOUT) ;
  if ( dgm ) {
    tm = new TextMessage (dgm) ;
    queue.enqueue (tm) ;
  }
}
```

```
receive (timeout) {
  time_waited = 0 ;
  msg = null ;
  while ( time_waited < timeout ) {
    msg = queue.dequeue_nb() ;
    if (msg)   break ;
    sleep (SLEEP_TIMEOUT) ;
    time_waited += SLEEP_TIMEOUT ;
  }
  return   msg ;
}
```

```
MsgProcessor loop:
while ( !quit ) {
  msg = receiver.
     receive (RECEIVE_TIMEOUT) ;
  if (msg) {
    // unmarshal-route-dispatch
  }
}
```

Fig 3-1: Server concurrent FSM. Notice the two concurrent sub-states nested into the Running state. Some notes with pseudo-code have also been attached to the diagram for further clarity.

Upon start up, the server enters into the *Initializing* state. Initialization is performed as already explained in the object model and two flows of control are allocated to run, respectively, the *Receiver* and the *MsgProcessor* processing loops.

Upon completion of the initialization phase, the server automatically transitions to the *Running* state, which is composed by two concurrent sub-states, the *Receiver loop* and the *MsgProcessor loop*. The *Receiver loop* is broken down into two states. In the *Listening* state the *Receiver* waits for incoming datagrams. When a datagram comes in (this is represented by the *newDgm* event), the Receiver puts the contained message on the queue, thus transitioning to the *Accessing queue* state. When both flows of control are in this state, they have to synchronize by acquiring and releasing the lock on the queue. After gaining exclusive access to the queue and adding the message, the *Receiver* transitions back to the *Listening* state. There is also a self transition to this latter state after a listening timeout has elapsed. This is to avoid the *Receiver* blocking forever on the

UDP socket. In fact, every time the *Listening* state is entered, the Boolean variable *quit* (initially set to *false*) is checked in order to see when it's time to exit the processing loop. If the *quit* variable holds *true*, then the *Receiver* exits its processing loop, releases the socket and waits for the *MsgProcessor* flow to join.

The *MsgProcessor* main activity (unmarshaling, routing and dispatching) is carried out in the *Processing* state. If no message is available to process, then the *MsgProcessor* tries to fetch one from the *Receiver*, by invoking the *receive* method. This causes a transition to the *Polling* state. The *receive* method regularly polls the queue by calling the *getMessage* method and putting to sleep the *MsgProcessor* for a given sleep timeout if no message is returned by the *getMessage* method (*noMsg* event, that causes an internal transition). When a message is available, this is returned to the *MsgProcessor* (*newMsg* event), which transitions back to the *Processing* state. If no message comes in within a timeout specified by the *MsgProcessor*, then the *receive* method gives up and returns a null value. This timeout is needed in order to allow the *MsgProcessor* to check when it's time to exit its processing loop. In fact, the *MsgProcessor* uses the same strategy as the *Receiver*: every time the *Processing* state is entered, the Boolean variable *quit* (initially set to *false*) is checked. If the *quit* variable holds *true*, then the *MsgProcessor* exits its processing loop and waits for the *Receiver* flow to join.

Upon dispatching of a shutdown request (represented by the *shutdown* event, which causes an internal transition in the *Running* super-state), the *CtrlSkeleton* (which runs in the *MsgProcessor* flow) will invoke its *shutdown* method, which will ask the *Receiver* and *MsgProcessor* objects to terminate execution. The *Receiver* and *MsgProcessor* both expose an *exit* method that sets the *quit* variable to *false*, causing the object to eventually leave its processing loop.

3.2. Allocation of processes and threads.

The Log Service makes use of the Perl 5.8 built-in threading environment. Specifically, two threads are allocated to the server-side and the client-side software needs to be able to allocate a different *Forwarder* object to each thread running in the client process, if the latter is a multi-threaded process.

The Perl threading environment that we're using is called *ithreads* (a short for interpreter threads). This choice represents a first attempt to explore Perl support to multi-threading and gain useful feedback in order to evaluate the possibility of using Perl threads on a larger scale within the OME Reference Implementation.

In the *ithreads* model each thread runs in its own Perl interpreter, all of its data is private and any data sharing must be explicit. In fact, when a new thread is created, all the data associated with the parent thread (the thread that spawned the new one) is copied to the new thread and can't be accessed from any other thread (including the parent thread) unless is marked as shared — through the *shared* attribute.
However, there are restrictions on what data may be shared. In fact, you can share scalars, arrays and hashes, but only simple values or references to shared variables may be assigned to shared array and hash elements. Moreover, even though references to shared variables can be passed among threads, it is not possible to share a blessed reference, which makes impossible to share objects directly.
Also notice that spawning a new thread after the parent thread has already accumulated a lot of data involves a considerable creation time overhead and subsequent expensive memory usage because of the cloning semantics of *ithreads*.

Back to the Log Service, how can the client-side software enforce just one *Forwarder* object per thread? The answer is not difficult. Recall that the *LogGateway* creates a proxy the first time the *getLogger* method is invoked and then recycles it in the next invocations. The proxy grabs the thread id at creation time by accessing the Perl threading API and stores that value into its *tid* field. The *Forwarder* object is also created and linked up to the proxy the first time *getLogger* is invoked. Now, for any given thread, only one of the following conditions will hold true whenever the *getLogger* method is called within that thread:

- There is no *LogProxy* object (and thus no *Forwarder* either). This means that the *getLogger* method is being invoked for the first time within the current thread. It also implies that the parent thread never called *getLogger* before spawning the child thread.
- There exists a *LogProxy* object (and thus a *Forwarder*), but the *getLogger* method has never been called before within the current thread. This is because the parent thread called *getLogger* before spawning the child. The *LogProxy* and *Forwarder* objects are copies of the original ones in the parent thread. Notice that, even though we have a copy of the *Forwarder* object, the file descriptor held by the object (recall that the *Forwarder* sets up the UDP socket at creation time) is exactly the same one — this is why we want to have one *Forwarder* per thread: we want to have a different UDP socket for each thread so that we don't need to synchronize access to a shared socket.
- There exists a *LogProxy* object (and thus a *Forwarder*) and the *getLogger* method has already been called within the current thread.

It turns out that in the first case we only have to create and link the *LogProxy* and

Forwarder objects and then return the proxy. This case is obviously detected by the absence of a link to a *LogProxy* instance.

The second condition is detected by the existence of the link to the proxy and by the fact that the current thread id is not the same as the one stored into the *tid* field of the proxy. In this second case we have to create a new *Forwarder* and re-link it to the copy of the proxy that we already have. We also have to set the *tid* field of the proxy to the id of the current thread, as it contains the id of the parent thread.

Subsequent invocations of the *getLogger* method within the same thread are detected by the existence of the link to the proxy and by the fact that the current thread id is the same as the one stored into the *tid* field of the proxy. In this case we only have to return the proxy instance. The *getLogger* algorithm is described by the following UML activity diagram:

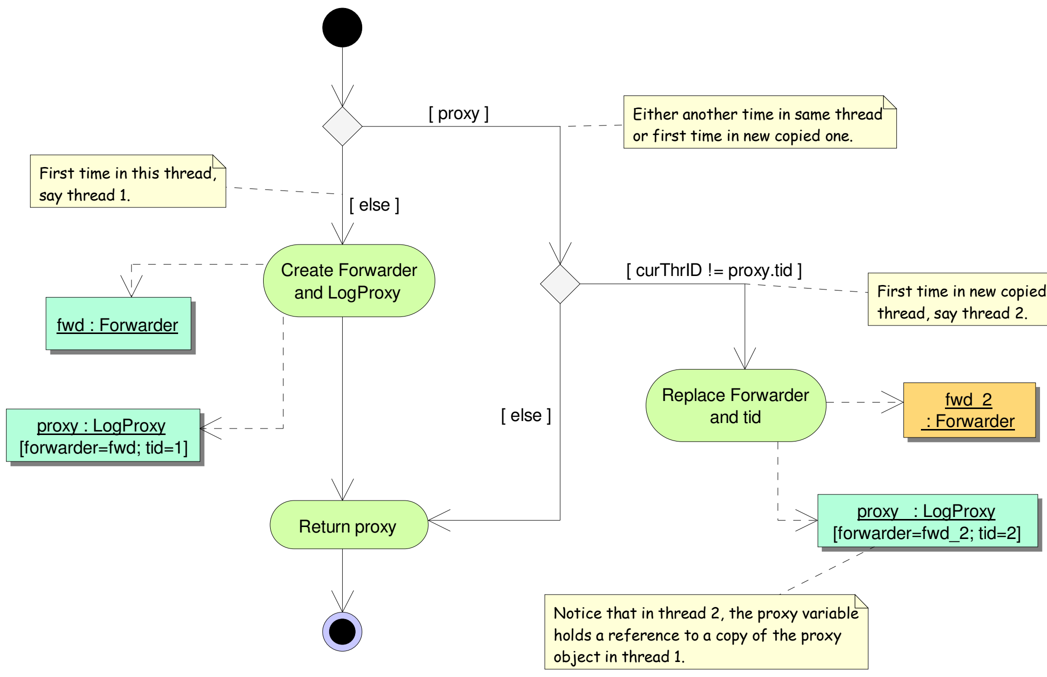

[proxy]
Either another time in same thread or first time in new copied one.
First time in this thread, say thread 1.
[else]
Create Forwarder and LogProxy
fwd : Forwarder
proxy : LogProxy [forwarder=fwd; tid=1]
[curThrID != proxy.tid]
First time in new copied thread, say thread 2.
Replace Forwarder and tid
fwd_2 : Forwarder
[else]
proxy : LogProxy [forwarder=fwd_2; tid=2]
Return proxy
Notice that in thread 2, the proxy variable holds a reference to a copy of the proxy object in thread 1.

Fig 3-2: The getLogger algorithm. Notice the object flow associated to the action states.

A final observation about client-side concurrency. Access to the *getLogger* method doesn't need to be synchronized. In fact, different threads will be invoking this method on their own private copy of the *LogGatweay*.

As you would expect, things get trickier on the server-side. We allocate two threads, one for the *Receiver* object and another for the *MsgProcessor* object. The *Receiver* object lives within the root thread. The *MsgProcessor* thread is spawned from the root thread. Here's the initialization code in the *LogService* class that allocates the threads and starts the processing loops:

```
sub start {
  my  $self = shift ;
  my  $receiver = new  OME::Log::Impl::Server::Receiver() ;
  my  $processor = threads->new( sub {
                my $p = new
                  OME::Log::Impl::Server::MsgProcessor($receiver) ;
                $p->run() ;
              } );
  $receiver->run() ;
  $processor->join() ;
  return ;
}
```

The *Receiver* and *MsgProcessor* constructors perform initialization as explained in the server-side dynamic model. As a convenience, we don't implement our own *MsgQueue* class, but we rely on the Perl utility class *Thread::Queue*. The behavior of this class is similar to that in the Monitor Object pattern [POSA2] — however we only need this class for its capability of serializing access to the buffered data. So the *Receiver* will instantiate a *Thread::Queue* rather than a *MsgQueue*.

As you can see in the code snippet above, the *MsgProcessor* is initialized with a reference to the *Receiver* object in the parent thread. However, what the *MsgProcessor* eventually gets is a copy of that object. In fact, the thread that runs the *MsgProcessor* object will initially contain a copy of all the data held by the parent thread at the time of spawning. This means that the *Thread::Queue* object and the Perl IO handle used to access the operating system UDP socket are copied too. However, the socket file descriptor can't be copied (obviously enough) and the two handles refer to the same operating system resource. The *LogSkeleton*, *Logger* and *CtrlSkeleton* objects are initialized within the new thread and so are private to this thread. The *MsgProcessor* object has to pass a reference to itself and to the *Receiver* object to the *Skeleton* static initialization method. However, the reference to the *Receiver* object that the *MsgProcessor* constructor passes along is a reference to the copied *Receiver* object. As a result, the *CtrlSkeleton* object gets linked to the right *MsgProcessor* object, but to the copy of the *Receiver*.

This means that the *quit* field of *Receiver* needs to be a shared variable. Otherwise, the *CtrlSkeleton* object wouldn't be able to set this field to *true* in the master *Receiver* object (the instance in the parent thread) when invoking the *exit* method on the copied object. Other data that need to be shared are the messages buffered on the queue. In fact, those messages are queued up by the *Receiver* on one side and fetched by the *MsgProcessor* on the other side — indirectly, through the *receive* method of *Receiver*. As the *Thread::Queue* class uses a shared array to buffer data internally, we won't have to worry about this. Nevertheless, we can't buffer *TextMessage* objects on the queue, because we may not share blessed references. As a result, the *Receiver* will queue up string messages as received from the UDP socket and the *receive* method (which runs within the *MsgProcessor* thread and removes messages from the queue) will take care of wrapping those strings into *TextMessage* objects and then returning those object to the *MsgProcessor*.

All the above might look a bit cumbersome, but it is unavoidable because of the cloning semantics of *ithreads*. The following UML object diagram is a snapshot of the server state after initialization and should help focus on the discussion.

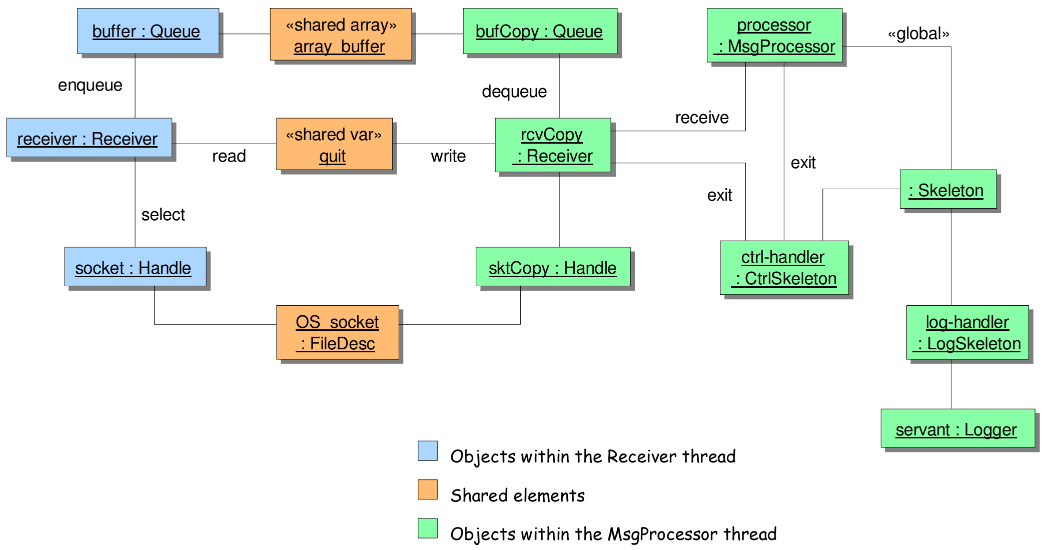


Fig 3-3: Snapshot of the server state after initialization. Different colors are used as a visual aid to distinguish objects that live in different threads and elements that are shared. The global stereotype means that Skeleton is visible to MsgProcessor because of static class visibility.

Looking at the above diagram, we can easily locate the elements that can be shared between the *Receiver* and *MsgProcessor* threads. Those elements may be accessed concurrently by the two threads and, for this reason, we need to decide on a synchronization policy for each element.

The shared array is managed by *Thread::Queue* and we won't have to worry about synchronization as this class already takes care of that — access is serialized internally by means of a lock.

Access to the *quit* shared variable of *Receiver* doesn't need to be coordinated between the two threads. In fact, this variable is periodically read by the master *Receiver* object (*receiver* in the above diagram) in the *Receiver* thread and is written once by the cloned *Receiver* object (*rcvCopy* in the above diagram) when the *CtrlSkelton* object (*ctrl-handler* in the above diagram) in the *MsgProcessor* thread invokes the *exit* method. It turns out that, even if read and write access may be overlapped, no harm can be done: the master *Receiver* will eventually read the value written by the cloned *Receiver* because the master cyclically checks the *quit* variable.

Finally, no action needs to be taken for synchronizing access to the operating system socket. The *exit* method is the only method of *Receiver* that is invoked within the *MsgProcessor* thread and that acts on the copied IO handle. This method closes the socket, but this won't damage the master *Receiver*. In fact, while the *MsgProcessor* thread is into the *exit* method (as a result of processing the shutdown request), the *Receiver* thread will be idling. This is because the shutdown request is the last request that can be received — the system administrator will have to make sure that all the other subsystems that use the Log Service have already terminated execution before sending the shutdown request to the Log Service.

3.3. IPC.

This section details the inter-process communication (IPC) in terms of communication protocols and means. We need to specify how client-server communication takes place within the Log Service.

We adopted a very easy communication protocol that we called OME Simple Log Protocol (OME-SLP). Communication rules are pretty straightforward: client-side software simply sends invocation requests to the server. No previous connection establishment, sessions or server responses are needed. Clients may send requests at any time and then carry on in their activity without having to collect a response from the server. The server processes requests in the same order as they come in and needs not issue responses.

Each request object is transformed into a machine-neutral data representation — which both server and clients agree upon — before it is sent to the server. This external data representation is ASCII encoded and line-based. Each line is terminated by the Internet line terminator, the ASCII byte sequence (0x0D, 0x0A) = (015, 012) = (13, 10).

Logging requests, represented by the *LogRecord* class, are encoded in the following way:

```
TYPE: 1 ¬
CONTEXT: χ ¬
TIMESTAMP: τ GMT ¬
PRIORITY: π ¬
MESSAGE: μ ¬
```

where ¬ is the Internet line terminator sequence and χ, τ, π, μ encode the information carried by *LogRecord* according to the following regex patterns:

```
χ    PID<(.*)> TID<(.*)> FILE<(.*)> LINE<(.*)> PKG<(.*)> SUB<(.*)>
τ    (.*)\/(.*)\/(.*) (.*):(.*):(.*)
π    (.*)
μ    (.*)
```

All of the above are self explanatory besides τ. It encodes a timestamp (relative to GMT+0) in the format:

```
dd/mm/yyyy hh:mm:ss
```

Server control requests, encoded by the *CtrlOp* class, are encoded in the following way:

```
TYPE: 0 ¬
OPID: δ ¬
```

where ¬ is the Internet line terminator sequence and δ encodes the information carried by *CtrlOp* according to the following regex pattern:

```
δ    (.*)
```

As already mentioned in several occasions, the transport mechanisms relies on UDP. The implementation is based on standard socket programming through the Perl socket functions that wrap the corresponding C system calls. This a routine programming task which doesn’t need to be detailed here. Just some quick notes though.

We explicitly set the size of the socket buffers both on the client and server side. This is mainly to achieve better reliability, as detailed in the Failure Model, and is done through the *setsockopt* Perl function, which is a wrapper to the corresponding C system call.

For every client-side socket, we should make sure that any pending message in the outgoing UDP buffer is eventually sent before the socket is closed. The *setsockopt* C call, allows you to do that. In fact, the *SO_LINGER* option flag tells the system to block the process on a *close()* until all unsent messages queued on the socket are sent or until a given timeout expires. If *SO_LINGER* is not specified, and *close()* is issued, the system handles the call in a way that allows the process to continue as quickly as possible, which means that the outgoing buffer may well be de-allocated before all pending messages are transmitted. The C function takes a *linger* structure to specify the state of the option and the linger interval. Unfortunately, the Perl documentation available for *setsockopt()*, doesn't tell you how to pass the *linger* structure. So we'll leave this out for now...

The *Forwarder* transmits data by calling the Perl *send* function on an unconnected socket. This eventually results in a call to the C *sendto* function, which will block until space is available at the sending socket if the outgoing buffer is full — this is the behavior of *sendto()* if the socket doesn’t have *O_NONBLOCK* set.

The *Receiver* uses the *select* function in order to perform asynchronous reads on the server socket. This is also a Perl wrapper function to the corresponding C system call. However a difference is that when using *select* from Perl, the helper macros *FD_SET*, *FD_CLR*, *FD_ZERO*, and *FD_ISSET* aren't available. Instead, Perl provides an assignable function *vec* which can be used to build the arguments to select. For example, the Perl statement

```
vec($rin, fileno(SKT), 1) = 1 ;
```

sets the bit corresponding to the file handle *SKT*, and

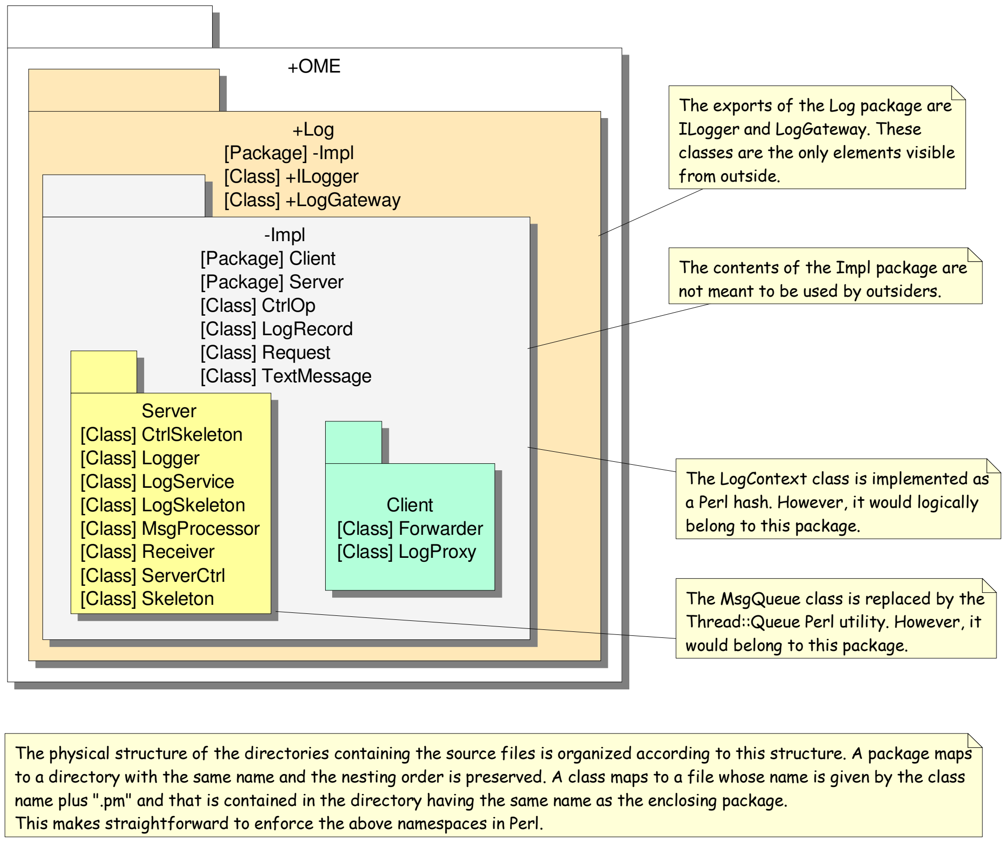
```
if ( vec($rin, fileno(SKT), 1) )
```

checks the same bit.

4. Mapping to Code.

The classes in the object model are mapped onto concrete Perl classes according to the most obvious paradigm: every class is a module and every module is contained in its own file, which has exactly the same name as the class plus the "*.pm*" extension.

Namespaces are arranged as shown by the packages in the following UML diagram (a package forms a namespace in UML):



The exports of the Log package are ILogger and LogGateway. These classes are the only elements visible from outside.

The contents of the Impl package are not meant to be used by outsiders.

The LogContext class is implemented as a Perl hash. However, it would logically belong to this package.

The MsgQueue class is replaced by the Thread::Queue Perl utility. However, it would belong to this package.

The physical structure of the directories containing the source files is organized according to this structure. A package maps to a directory with the same name and the nesting order is preserved. A class maps to a file whose name is given by the class name plus ".pm" and that is contained in the directory having the same name as the enclosing package.
This makes straightforward to enforce the above namespaces in Perl.

Fig 4-1: Logical namespaces. Notice the visibility symbols used to denote public (+) and private (-) elements.

The physical structure of the directories containing the source files is organized according to this structure. A package maps to a directory with the same name and the nesting order is preserved. A class file is contained in the directory having the same name as the enclosing package. This makes straightforward to enforce the above namespaces in Perl.

As a final remark, notice the absence of *MsgQueue* and *LogContext*. The former is replaced by the *Thread::Queue* class, as already stated in the process model. The latter is implemented as a Perl hash.

5. Deployment.

In the current deployment configuration, both the client-side and the server-side of the Log Service are deployed on the same processing node. Client and server make use of the network loopback interface and the server port is *12345*. Thus, we have distribution across processes, but those processes have to be located in the same machine. This will be changed if we need the clients to be located in different machines. However, for the time being, this is not needed.

The server is started through the *LogService.pl* script, that has to be run passing *start* as a parameter on the command line. To shut the server down, just run the same script, but pass *stop* as a parameter. Client classes access the service by importing the *OME::Log::LogGateway* module.

All the Log Service Perl modules are contained into the *Log* directory that has to be placed under the main *OME* directory into the base directory of the Perl development tree. The Perl script *Makefile.pl* into the base directory of the Perl development tree will write an appropriate make file so that all the modules in the base directory can be installed into the proper system locations and can be therefore available to the Perl interpreter.

The *LogService.pl* script also goes into the base directory of the Perl development tree, along with the *OME-LS.conf* file. This latter file is the configuration file that is used to fine-tune the logging behavior and has to be **<in the same directory as the *LogService.pl* script?>**.

<Talk about config file and give examples to show how to fine-tune behavior>

The Log Service depends on the Log4perl library, that needs to be installed before starting the service — this library is only required by the server side, so in the case of distribution across different machines, the client machines won‘t need it. The target platform is any platform that supports Perl. However, as we make use of *ithreads* (the new Perl threading model), the Perl interpreter has to be 5.8 or greater and it must have been compiled with *ithreads* support. This should be the default in most cases, however run *perl -V* and make sure that the *Platform* section contains *useithreads=define*. Notice that your Perl interpreter mustn't be configured to support both the old threading model (5.005 model, which is highly deprecated anyway) and *ithreads*. If you also see *use5005threads=define* in the *Platform* section, then you should reconfigure the interpreter in order to support only *ithreads*. Please refer to the *ithreads* man page for further information.

6. Failure Model.

This section is devoted to seeking for ways in which failures can occur and to analyzing such failures. The goal is to understand the effects and impact of failure in order to determine a failure handling policy.

We grouped failures according to the layer where they arise: *Transport*, *Distribution* or *Application*.

The *Transport* layer has to manage communication over UDP. In the general case of a sender process connected to a receiver process across a network, the message transmission can suffer from the following omission failures [HT94]:

- *Send-omission failure*: A message is lost between the sending process and the outgoing operating system buffer, due to buffer overflow.
- *Channel omission failure*: A message is lost between the sending process and the receiving process, because of a network transmission error (detected by a checksum) or because of buffer overflow at any of the network nodes between the sender and the receiver.
- *Receive-omission failure*: A message is lost between the receiving process and the incoming operating system buffer, due to buffer overflow.

Moreover, messages can arrive to the receiver process out of sender order, generally because of different paths followed by messages across the network nodes from the sender to the receiver.

In our case, both the sending and receiving process are located in the same machine and connected through the loopback interface. Thus, the operating system simply moves UDP datagrams from the client socket to the server socket, in a single step. As a consequence, we won‘t have to deal with channel omission failures. Also, datagrams cannot be reordered and, as a log message is forced into an UDP datagram (*TextMessage* truncates too long logs), log messages arrive to the server in the sender order.

Buffer space is not a problem on the client-side. In fact, the *Forwarder* will block until space is available at the sending socket if the outgoing buffer is full — this has already been discussed in the Process Model. This means no send-omission failures are possible.

Receive-omission failures are dealt with a large socket buffer on the server-side.
… (intensive testing proved this hypothesis correct on a large set of possible configurations on different systems where the Log Service is likely to be run, please refer to the test document.)

duplicated, or lost (the last is an idealization, since in actual systems the OS could drop them for lack of space).
<
No data fragm => no ordering problems

A highly reliable comm cannel could be easily built…
A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgments.
However, send/recv omission failures are very rare (show and discuss graph) and for this reason it's worth trading off improved performance (no wait for server reply) over msg lost.
>

The *Transport* layer can suffer from failures related to the interaction with the UDP API.

<Say that communication channel is reliable (give CDK definition), however some minor issues regarding access to sockets have to be considered. Those are described in the following table.>

The following table summarizes the failures that can arise in the *Transport* layer, their effects and impact as well as the relative failure handling policy.

Failure	*Context*	*Effects / Impact*	*Handling*
1. Socket creation.	*Forwarder* creation by *LogGateway*.	Client objects can't access the service. Impact depends on client objects logging policy.	Exception is thrown and client objects decide whether or not to carry on without logging support.
2. Socket creation.	*Forwarder* creation by *ServerCtrl*.	*LogService* script can't send shutdown message. Administrator might have to kill server process.	Exception is thrown and *LogService* script exits. Administrator retries shutdown.
3. Socket creation.	*Receiver* creation at server start up.	No communication end point available. The Log Service can't be started.	Exception is thrown and *LogService* script exits. Administrator retries start up.

4. Socket binding.	*Receiver* creation at server start up.	Network address and port can't be bound to socket. The Log Service can't be started.	Exception is thrown and *LogService* script exits. Administrator retries start up.
5. Message sending.	*Forwarder* sends a message on its socket in client process.	Log message is not received (and thus recorded) by the server.	No action.
6. Message sending.	*Forwarder* sends message on behalf of *ServerCtrl* in process running *LogService* script.	*LogService* script can't send shutdown message. Administrator might have to kill server process.	No action. Administrator retries shutdown if server is still on.
7. Message reception.	*Receiver* invokes *recv()* to collect message from OS buffer.	Message is lost. If logging request, then record is lost. If shutdown request, administrator might have to kill server process.	Error message is output on server's *STDERR* and server carries on with its normal activity.
8. Socket release.	*Forwarder* garbage collection in client process.	Client process will still hold an unused socket. Increased system resources consumption.	No action.
9. Socket release.	*Forwarder* garbage collection in process running *LogService* script.	Process running *LogService* script will exit shortly after anyway (socket will then be released).	No action.
10. Socket release.	*Receiver::exit()* is invoked.	Server process will exit shortly after anyway (socket will then be released).	No action.

Notice that for … we take no action. This is because the possibility that the socket file descriptor fails to be released is very low. This means that the number of non-closed sockets in any client process will be zero most of times or, at worst, always close to zero. Thus, the resource wasting can be considered irrelevant.

Same consideration for message sending.

6. No exception, admin sees sever still on and retries shutdown
7. Shouldn't happen b/c recv called after select.

General policy is that if likelihood is very low, then better keep design simple and manual recovery instead.

Should necessity arise, we can provide a more robust failure handling.

< TO BE PUT into final considerations?

Some assumptions have been made in order to simplify design, decrease network communication overhead and minimize the time that calls to local proxies need to complete. As a result, the current implementation of the Log Service won't deal with:

- *Omission failures.*
- *Arbitrary failures.*
- *Data fragmentation.*

Omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do [CDK01]. The Log Service transport layer doesn't check for server availability nor does it enforce reliable communication (UDP doesn't guarantee message delivery). As a result, the Log Service may fail to record a log message for any of the following reasons:

1. The server side of

-- A log record will fit into UPD datagram (1Kb or 8Kb?), no fragmentation is needed. Actually cut off on client side if msg > MAX_SIZE (=1 or 8Kb, depending on what found out). Mention that msgs longer than 4kb = MAX_SIZE will be truncated. However, a log string is usually in the range 50-500bytes.

-- Now explain that assuming LAN => lost msgs possibility ~0 (what if buffer overflow?=>however could be dealt with fast dispatcher and no bounded queue), but still electrical noise on network (if !checksum => datagram dropped), to eliminate this too => same machine. Notice that would be easy to make it run on LAN if added some ACK.

However design is such that future modifications due to accommodate… will only affect the implementation of the transport layer — namely the Forwarder and Receiver classes. Also notice that as those classes completely encapsulate the lower level communication channel and end-points, it is relatively easy to replace the UDP transport with TCP.

Notice: the shutdown message has to be the last one sent to the server => last one on the queue. How do we guarantee that? Log Service is shut after all other services (this doesn‘t ensure shutdown last one, coz other messages might still be on their way to the server). Messages that might come in between shutdown reception and socket release are ignored. So admin must allow a reasonable amount of time after services shutdown before issuing Log Service shutdown.

Recall what said in 3.2 (by the end): …This is because the shutdown request is the last request that can be received — the system administrator will have to make sure that all the other subsystems that use the Log Service have already terminated execution before sending the shutdown request to the Log Service.

>

<Put the above into a separate section: FAILURE MODEL and refer to this section within the above discussion>

References

[BBC+00] F. Bachmann, L. Bass, J. Carriere, P. Clements, D. Garlan, J. Ivers R. Nord, R. Little:
Software Architecture Documentation in Practice: Documenting Architectural Layers
Special Report CMU/SEI-2000-SR-004, available at
http://www.sei.cmu.edu/publications/documents/00.reports/00sr004.html

[BRJ00] G. Booch, J. Rumbaugh, I. Jacobson:
The Unified Modeling Language User Guide
Addison-Wesley, 2000

[CDK01] G. Coulouris, J. Dollimore, T. Kindberg:
Distributed Systems — Concepts and Design
Addison-Wesley, 2001

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides:
Design Patterns — Elements of Reusable Object-Oriented Software
Addison-Wesley, 1995

[HT94] V. Hadzilacos, S. Toueg:
A Modular Approach to Fault-tolerant Broadcasts and Related Problems
Technical report, Dept. of CS, University of Toronto, 1994

[Larm01] C. Larman:

Protected Variations: The Importance of Being Closed
Article on IEEE Software, May/June, 2001, available from
http://www.craiglarman.com/

[L4J03] C. Gülcü, et al.:
The Log4j Project
Project page at http://jakarta.apache.org/log4j/

[L4P03] M. Schilli, K. Goess:
The Log4perl Project — Log4j for Perl
Project page at http://log4perl.sourceforge.net/

[OMG01] Object Management Group:
Unified Modeling Language Specification v1.4
Available from http://www.uml.org/

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal:
Pattern-Oriented Software Architecture —A System of Patterns
John Wiley & Sons, 1996

[POSA2] D. Schmidt, M. Stal, , H. Rohnert, F. Buschmann:
Pattern-Oriented Software Architecture —
Patterns for Concurrent and Networked Objects
John Wiley & Sons, 2001