Maintenance Documentation

Server

The i-scream server is the central part of the i-scream Distributed Central Monitoring System. It collects, processes and stores data. It also generates alerts as necessary.

Revision History

24/03/01	Initial creation, not yet complete				
		Committed by:	tdb1	Verified by:	ajm4
				Date:	25/03/01
25/03/01	More detail in filter/filtermanager areas				
		Committed by:	ajm4	Verified by:	tdb1
				Date:	26/03/01
26/03/01	Completed document				
		Committed by:	tdb1	Verified by:	ajm4
				Date:	27/03/01
27/03/01	More detail in the configuration system				
		Committed by:	ajm4	Verified by:	tdb1
				Date:	28/03/01
28/03/01	Added system diagrams to the end of the document				
		Committed by:	ajm4	Verified by:	tdb1
				Date:	28/03/01

Introduction

The i-scream server is the central part of the i-scream Distributed Central Monitoring System. It links together the hosts, clients, and webpages, and provides a central point of configuration for most of the system.

The i-scream server has been built around a component architecture, with each component communicating using CORBA[1]. This allow each component to be run separately, and even on different machines, thus breaking a large application into something much more manageable.

The whole server is written in Java, version 1.2 or later, and uses the JacORB[2] CORBA implementation. It is self-contained, and can be run without installing anything other than Java. It should run on both Unix/Linux systems and Windows, although compiling must be done under Unix/Linux due to the Makefile setup.

This document aims to discuss each section of the server in a manner that would allow other developers to maintain it. It should also be noted that the code itself contains extensive javadoc, which is also viewable on the i-scream website.

[1] http://www.corba.org/
[2] http://jacorb.inf.fu-berlin.de/

Overview

Server Architecture

The server itself is broken into a series of distinct components, which each component serving a specific role in the operation of the system. These components link together using CORBA to transfer data around.

There are a collection of classes that are used across all components, namely a Reference Manager (aids CORBA communication), a ConfigurationProxy (cuts down on CORBA communication), and a ComponentManager (handles startup).

The Core component provides services to the rest of the server, namely logging and configuration. The centralised point of logging allows all of the components to log to the same destination, be this a file or the screen. One of the key features of the server is that it is centralised, yet distributed over components. To help achieve this the configuration for the entire system is centralised into the Core component, which interacts with a single selection of files. This configuration is then available to any component that requests it.

The first component a host come into contact with is the Filter Manager. The role of this component has changed since it's conception, and now just dishes out Filter addresses to hosts. It does this through querying the Filter configured for the requesting host, and passes the given details back to the host.

The Filter component is the entry point of host data to the system. It filters invalid XML data, and data that does not conform to the specification, and then passes it on. Filters may be chained up in a hierarchical structure, but ultimately the data must reach the Root Filter.

The Root Filter component is where data is distributed to the database and clients. Although it masquerades as a Filter, it doesn't actually accept connections from hosts, only from other Filters.

The Database Interface simply inserts data into a mySQL[3] database, after time stamping it. It hooks directly to the Root Filter.

The Client Interface acts as a connection point for clients, and provides them with a stream of data. It handles both TCP and CORBA connections.

The Local Client is really a client, and connects as such. It handles all the alerting behaviour of the server, and is in fact a complex program. It normally requires more resources than the rest of the server. However, as only one of these clients usually exists it is bundled with the server.

[3] http://www.mysql.com/

Central Services

This section looks at some of the central services. These are typically involved in all components, and are therefore discussed prior to going into detail about each components architecture.

Component Manager

Overview

The Component Manager is the program that is loaded first, and handles starting up other components in the system. It also has extra behaviour to run the required dependency checks to ensure the system starts in the right order.

Purpose

It very quickly became clear that we needed something external to the component architecture to allow the system to be started up nicely. Originally the Core component handled starting up, but this tied down the way in which the system could be distributed.

Using this new architecture, a non-component program to start things up, we can easily distributed components around different JVMs and even different machines. This flexibility is one of the key features in the i-scream server.

Detailed Analysis

The component manager starts by reading in its default properties file. This specifies the list of components that should be loaded into this JVM. It also specifies other defaults, such as where the system configuration is, and which logger to use – all information which needs to be acquired prior to starting up the Core component (and thus the ConfigurationManager).

Once it has the component list it attempts to construct all of these components. All the components implement a Component interface, and can therefore be treated the same way. Construction of a component should not cause anything to be started; otherwise it bypasses the dependency checks.

When all the components are loaded the component manager goes into its startup routine, which should end in all the components being active and ready. This is essentially a loop in which each component goes through two stages. Firstly a dependency check is carried out by the component itself. In this stage the component does all the necessary checks to see if components it depends on are already started. If this returns a positive response, the component manager proceeds to the next stage of starting the component. This process is repeated until all components have been attempted. If, after this, there are some components that failed the dependency check a delay is inserted then the process repeated, this time with only the failed components. This continues until there are no failed components.

Left to it’s own devices this means that all components will eventually be started, in the correct and required order. This can happen across multiple JVMs or machines, as the dependency checking is carried out through a CORBA naming service lookup and is thus capable of finding components outside of its startup environment.

The one area not covered by this architecture is stopping and/or restarting of components. It is not yet possible to do this, although both the Component interface and the ComponentManager could be extended to allow this.

Reference Manager

Overview

The Reference Manager is a singleton class that helps to wrap the CORBA method calls away from the rest of the system. It also has methods to enable components to quickly obtain references to the Logger and Configuration Manager (explained in the Core component later on).

Purpose

It quickly became apparent that a lot of the code to deal with CORBA communications was being duplicated across various components of the server. It was also obvious that in some cases this was not being done as well as it should have been. In an aim to tidy this up the Reference Manager was written, and introduced as a singleton throughout the system. Being a singleton any class can get a reference to a single instance of it (on a per JVM basis).

It has methods that enable some lengthy CORBA operations to be completed in a single command, and has complete error checking built in.

Detailed Analysis

Here some of the main features will be looked at.

getCORBARef(String name)

This method is one of the most used in the Reference Manager. It allows a CORBA object reference to be retrieved from the naming service. This reference would then need to be narrowed by the appropriate helper class. Error handling is done internally, with fatal errors shutting down the system in a more presentable manner than the usual exception.

bindToOrb(org.omg.PortableServer.Servant objRef, String name)

This method is used by servants throughout the system to bind themselves to the ORB, and to the naming service. This must be done before a servant can be addressed by any other component in the system. Prior to calling this method a servant would normally obtain an object reference to itself.

getLogger()

This method is used by practically every class as it returns a reference to the system logger provided by the Core component. Every class should log something, and this makes obtaining a Logger reference a lot easier. This method deals with all the narrowing, and thus can be used directly (i.e. getLogger().write.....).

Other Methods

There are other methods made available by the Reference Manager, but these are used less. Amongst them is getCM() which returns a reference to the Configuration Manager, getNS() which returns a reference to the Naming Service, and getRootPOA() which returns a reference to the Root POA.

It is recommended that this class be used wherever possible for doing CORBA related communications.

Configuration Proxy

Overview

The Configuration Proxy acts as a singleton class, present in every component JVM, and performs caching of configuration requests on behalf of other classes. For more information on how the configuration works, please see the Configuration documentation.

Purpose

One of the main ideas from the start was to allow a dynamic reconfiguration of the whole system. This, however, isn't always possible without a lot of extra work, but the Configuration Proxy is one step in that direction.

The Configuration Proxy is a singleton class, and as such only exists once in each JVM. It handles all configuration lookups on behalf of other classes in the same JVM, and caches responses. The intention is that instead of retrieving and storing configuration other classes will use the Configuration Proxy for every lookup. As they reside in the same JVM, this isn't too expensive, at least when compared to doing the same over CORBA.

The Configuration Proxy takes each request, performs the lookup to the main configuration system, and then returns the result to the caller. At the same time it also caches this response, and returns it to future requests for the same property. At the same time there is a periodic checking running, which verifies if any configuration has changed, and drops any out of date configuration it has.

Detailed Analysis

The Configuration Proxy is broken down into two main sections; the serving requests section, and the check for expired data section. These will be looked at here.

Serving Requests

The main method of serving requests to the caller is through the getProperty method. This method allows the caller to request any property from any configuration. From the callers point of view this is a very simple procedure, although dealing with the exception thrown if no property is found is not. The idea behind this exception was that it ensured the calling class took precautions in the case of the property not being present, something which left unchecked could crash a component.

The getProperty method makes use of a ConfigurationCache object to cache information for a single configuration. Every time a property is requested it is placed in the relevant cache (or a new one is created) and then returned to the caller. It is important to note the care taken over null values returned; they must be placed in the cache and then an exception thrown. If the exception is thrown too soon the null won't be cached and the next request will also invoke a connection to the system configuration.

The ConfigurationCache is a local cache of a remote Configuration[4] object. It holds a reference to the remote Configuration object, for future lookups, and a HashMap of previously requested properties. When a request is made a check is made in the HashMap prior to asking the Configuration object for it.

It should be noted that the caching method employed, through the ConfigurationCache, holds a reference to a Configuration object at the ConfigurationManager in the Core component. This is not a problem, as long as the Configuration object is disconnected when finished with.

There are two other methods for returning the file list and the last modified date for a given configuration. All these do is pass the request on to the cached Configuration object.

[4] A Configuration object is a CORBA servant residing in the Core component

Expiry Checking

The Configuration Proxy would be rather useless if it didn't periodically check for new configuration, and thus implementing the dynamic side of the configuration. A cycle is running even N seconds, where N is configurable, which checks for out of date configurations.

This is done by asking a Configuration object, retrieved from a ConfigurationCache, for it's file list and last modified date. The ConfigurationManager, in the Core, can be provided with these details and asked to run a check. If the check reveals that the Configuration object is out of date it is removed.

This removal is done by first disconnecting the old Configuration object. This step is imperative, otherwise the Configuration object will never been garbage collection. The next stage is to get a reference to the new Configuration object, and place that in the ConfigurationCache, then to replace the old cached properties with a new empty HashMap.

When the next request comes in, it will hit the empty HashMap causing a lookup to the new Configuration object. This will get the latest configuration.

Server Components

This section looks at the individual components that construct the server. Each one is separately constructed, and can thus be discussed independently from the others.

Core

Overview

The Core component is the central point of the server in terms of construction, although not in terms of data flow; no host data ever passes through it. It provides to services to the i-scream server, the central configuration system, and the central logging facilities. It has no dependency on the rest of the system, although nearly every other component depends on it.

Purpose

This component was on of the first written, as it was the corner stone of the entire server. Every component would depend on it for configuration, and for logging. This ties in with the original specification for a “Centralised monitoring system”, with this component fulfilling the centralised requirement.

Using this component we can keep all configuration, for both the server and external programs such as hosts, in a central tree of files. This makes updating configuration a relatively straight forward, and solves a lot of problems found when having configuration distributed over the network. This component also provides a central point of logging, meaning that all messages about all the components states can be sent to the same file or console, regardless of their origin in the server.

Detailed Analysis

The Core is broken into two very distinct sections, the configuration and logging. Whilst both are completely separate, they do both need each other, and led quite soon to a classic “chicken and the egg” problem, do you allow the configuration system up first so the logger can obtain its configuration from it, or do you bring the logger up first so the configuration system can log important startup messages. In the end it was decided to go with the latter, as the configuration system played a more crucial role in the operation of the whole system.

Configuration

The configuration system appears to be rather simple from an external point of view, but underneath it has many complex and powerful features. These include separate configuration files for different parts of the system, configuration grouping, and inherited configuration. All of this is boiled down into the single ConfigurationManager class.

For other components the configuration system is used by sending a request for a configuration name to the ConfigurationManager, to which a Configuration object is returned. This Configuration object can then be queried for specific properties found in the requested configuration. However, in the last weeks of development a ConfigurationProxy was introduced, which sits between components and this architecture, and it is recommended this be used in preference. See the relevant section elsewhere in this document for details.

When a particular configuration is requested, the ConfigurationManager must determine the specific file list for it. This involves finding out which groups the configuration is a member of, allowing pattern matching on the names. The group matching can be done either by whole name, or a wildcard name that matches the configurations requested. With wildcard matching it is possible to place a ‘*’ anywhere in the name to match more than one configuration, this option is extended for hosts which can not only be matched by hostname, but also by their IP address (thus allowing wildcarding by subnet). For examples and possible uses of this, please refer to the Configuration section of the user guide.

Once a configurations group membership has been determined it then looks up to see which configuration files are assigned to those groups. When it has a list of configuration file names for groups, it extends this list by searching the files for include statements (which allow further configuration files to be included). This file list is a priority ordered list, in that the top one is used first, and if the requested property is not found, the second one is tried, and so on. At the end of the list resides the main configuration file, usually system.conf. If no files, groups, or specific configuration is found, then the main configuration is the only configuration returned. This then means that the main configuration should be used as a place to locate all system wide and default configuration.

Once this file list is built, it is given to a new Configuration object, which builds a java Properties class on top of it, using the built-in features to allow multiple files to be used in priority order. This Properties class is then made available to the original configuration requester, through the accessors on the Configuration object.

The final stage of this process is to remember to detach the Configuration object. This is done through the disconnect method, which releases the object from the CORBA POA. Typically the ConfigurationProxy handles this when it is finished with a configuration.

This configuration system allows potentially complex configurations to be built from a rather simple selection of files. For more complete detail, the configuration section of the server user guide provides information about the various options available and examples of how flexible configurations can be built.

Logging

The concept of the Logging system is quite simple. A single CORBA servant waits for logging requests from other parts of the system, and when it receives them it passes it on to an underlying implementation to actually be logged. The underlying implementation is very separate from the rest of the system and allows different logger classes to be written easily to suite anyone's needs (e.g. one that might interact with another logging system).

The first thing done is to read the desired underlying logger implementation name from the default properties file. The main two are ScreenLogger and FileLogger. Once this name is acquired, reflection is used to instantiate the class, which should implement the LoggerImpl interface. Thus any new implementations could easily be written and dropped in, as long as they implement the LoggerImpl interface.

With the underlying implementation in hand we start up the LoggerServant and pass it a reference to this implementation. Finally we bind it to the naming service, and it's available to the rest of the system.

Any component can now obtain a reference to the LoggerServant over CORBA and call the write method, with a simple textual message. This information will be presented nicely, using the FormatName class, and dumped to the logging implementation, which should send it to the appropriate destination.

This design allows the logging to be implemented easily, and in sensible stages. It is also flexible enough to enable further logging mechanisms to be easily implemented and be plugged straight in.

Filter Manager

Overview

The Filter Manager handles initial configuration of hosts, and assignments to Filters. Despite it's name, it doesn't actually manage Filter's, although this was originally its role.

Purpose

One of the main things we wanted to do with hosts was centralise the configuration, as they are the part of the system that the user will least want to maintain, primarily because there are potentially lots of them. We therefore decided to use the central configuration in the server, provided by the Core component. However, hosts are designed to be as lightweight as possible so as to reduce load on the machine on which they are running. This means hosts do not have knowledge to engage in heavy weight CORBA communications, so a Filter Manager was introduced to allow the hosts to negotiate with the server.

It should be noted that despite the name, the Filter Manager looks after hosts, and assigns them to Filters. It was originally designed to manage Filter's, but instead it became apparent that the CORBA naming service could provide a method of Filter registration and so the Filter Manager now interacts with this to determine a Filter's status.

When a host is started all it needs to know is the address of the machine the Filter Manager is running on, and a port number. It then connects to this port using the Host-Server protocol, as laid out in the protocol specification. The Filter Manager then negotiates with other server components on the host's behalf.

Detailed Analysis

The Filter Manager is the least complex of components, with a mere three classes. All it actually needs to do is listen for connections and fire off a handler to deal with each new connection – thus allowing multiple concurrent connections.

The HostListener class listens for connections and launches a new HostInit class, as a thread, to deal with every incoming host. The HostInit class conforms to Host Connection protocol as described in the protocol specification document. One of the key pieces of information exchanged in this negotiation is the fully qualified domain name (FQDN) that the server sees the host as connecting from. The host will then use this information in all host identification. The reason for this is twofold. Firstly, it was difficult on some platforms for the host to easily obtain the FQDN as opposed to just the hostname. Secondly, it ensures that both the server and the host are using the same name for identification, as many hosts (particularly ones with multiple network interfaces) may report different FQDN's to the one the server can look up.

The majority of the information sent during this communication deals with configuration. The host can get configuration for itself from the server by simply requesting the properties it requires. The HostInit class requests the required information from the configuration system, and passes it back to the host. The host also gets sent the file list and last modified timestamp of the configuration in use. These are required in the heartbeat protocol, and allow a host to check when its own configuration has changed rather than expect the server to maintain this information.

The last, and probably most important, data sent is the address and port number for the machine that the Filter is running on. A host can communicate with any Filter, and it is up to the Filter Manager to assign the correct one. This is done by looking up the name (or list of names) of Filters that have been configured for the host. The Filter Manager then processes this list in the order it appears. It first resolves a reference to a Filter; it then asks the Filter what host and port it is currently running on. If all the checks are successful, it returns the information to the host, if however it fails at any stage the Filter Manager moves on to check the next Filter in the list. If the end of the Filter list is reached and no configured Filters have been found, the Filter Manager informs the host and communication is ended. The host can then choose what it should do next, more often than not it would be best to wait for a timeout period and try again.

Once this negotiation has been completed successfully, the host will then enter a data sending and heartbeat cycle.

All of this ensures the Filter Manager is pretty stateless, and it is therefore feasible to run more than one in a single system.

Filter

Overview

The Filter component is the entry point of data to the server from hosts. It has filtering capabilities, and has features to allow Filter's to be chained together.

Purpose

The purpose of the Filter is to distribute the load of incoming data, and allow bad data to be filtered before it enters the central part of the server. The design of a Filter allows distribution across a network, thus localising UDP traffic to a single network segment. Each Filter is given a 'parent' Filter to which it must send all data, and this can be another Filter, or the Root Filter. Ultimately the hierarchical tree must reach the single Root Filter. Each Filter has a built in queuing mechanism, which allows the system to cope with large bursts of data.

Detailed Analysis

This section details the various parts of the Filter, and how they work.

Receiving Data

The Filter has three points for entry of data. These allow it to receive communication over different mechanisms.

The bulk of data sent to a Filter is UDP statistics from hosts. This data is collected using the UDPReader class, which is started by the main component class FilterMain. It loops round receiving byte arrays, converts them to Strings, and adds them to the processing queue.

The host heartbeat protocol also requires the Filter to receive TCP communication. The TCPReader and TCPReaderInit classes handle this. As with most TCP communications it is necessary to handle more than one connect concurrently, which requires threading. The TCPReader class listens for connections and launches a TCPReaderInit thread to deal with each connection it receives. The TCPReaderInit class communicates with the host using the Host Heartbeat protocol, as laid out in the protocol specification. One of the key events at this point is the host checking to see if its configuration has changed. A configuration change indicates to the host that it should reconfigure itself, however it does not do this with the Filter, instead it completes the heartbeat protocol and then returns to the FilterManager to re-initialise itself. Once a TCP communication has been completed, a heartbeat packet is generated and queued. This heartbeat packet indicates to the system that a host is still alive. The need for this arises because of the lack of guaranteed delivery of a UDP data packet. A host may still be alive, but its UDP data may be lost, so a periodic pro-active connection is made by the host to confirm to the i-scream system that it is still alive. A heartbeat from a host also fire Service Checks against the machine the host is running on, although these will be discussed later it is important to note that they are fired here.

The third and final method of receiving data is over CORBA. This allows other Filters to deliver packets of data directly into this Filter and thus enable chaining of Filters. The XML data is received as a String by the FilterServant, and again queued up.

Data Processing

All data received by a Filter must be processed. This is a multi stage process, and could result in the data being rejected by the Filter.

First off the data is pulled from the queue by the FilterThread, and is then turned into our custom XMLPacket object. This object is passed in to the PluginFilterManager to be checked

for filtering – this will be explained later. If the data is not filtered, it is passed on to the 'parent' Filter, which may be another Filter or the Root Filter.

This process ensures that any data that does not conform to the XML or i-scream standards is rejected. Rejection at this early stage is preferable, as it cuts down the load on the central parts of the i-scream server.

Filtering

One of the key features of a Filter, apart from data collection, is filtering. This is done through maintaining a set of 'filter plugins', each performing a specific filtering task. These plugins conform to an interface (PluginFilter), and are activated in the server configuration. Each plugin is given a reference to an XMLPacket, which is must check. It can then return either true or false, with false indicating the packet has been rejected by that plugin. If any of the plugins return false, the checks are immediately halted and the data is discarded.

These plugins will be individually explained in more detail later on.

Service Checks

Another feature of a Filter is the ability to run service checks against the machines a host is running on, as mentioned before, these are fired when a host carries out a TCP heartbeat with the Filter. These checks are only basic and simply verify if a service is active. As an example the HTTP service check will check to see if a webserver is responding on the host. Each host can have an individual set of checks run, defined in the configuration. The result of these checks is encapsulated in XML and added to the heartbeat packet (generated by TCPReaderInit) before it is sent.

The service checks themselves are organised in the same manner as the plugin filters, in that they conform to an interface and are managed by the PluginServiceCheckManager. This manager is told to run checks on a specific hostname, and it returns XML data indicating the result. Internally it keeps service check plugins active rather than creating them each time, and handles checking the configuration for each host.

Again, these plugins will be individually discussed later on.

Root Filter

Overview

The Root Filter masquerades as a Filter although it does contain some key differences. It sits at the root of the Filter Hierarchy and acts as a final collection point for data.

Purpose

The Root Filter serves two purposes. Firstly it conforms to the IDL specification for a Filter (i.e. it can be contacted over CORBA as a Filter), thus ensuring all other Filters can send their data on to it with ease as they do not need to be aware that they are talking to anything other than a normal Filter. It doesn't, however, accept TCP or UDP connections from hosts. This means it does not need to have any filtering facilities, as other Filters will have done this upstream.

The second purpose of the Root Filter is to distribute data to the client interfaces, which at present there are only two of – one for real time clients (eg. Conient) and one for the database (though more could be written if the need arose).

Detailed Analysis

Here we look at how the Root Filter performs its various tasks.

Data Collection

The Root Filter only collects data from other Filters, and as such only needs a CORBA listening mechanism. This is handled by the RootFilterServant class, which simply receives data and adds it to a queue. No processing is actually done, just queuing at this stage.

Passing data on

The Root Filter must pass data on to the client interfaces, and this is done through the use of the CIWrapper class. This wrapper class evolved because originally data was pushed straight to the client interfaces upon receipt, but after the introduction of a queue this wasn't directly possible. To solve this, a wrapper class was written to pull data from the queue, and push it to a client interface. The CIWrapper class is started in the main component class, RootFilter, and is assigned to an individual client interface – thus more than one may be started.

It should be noted that the client interfaces were designed to be pluggable – i.e. they conformed to an IDL interface, and thus more could easily be added. Although this was implemented, it was never really made into a main feature. It works by the names of the client interfaces being listed in the configuration. The Root Filter then attempts to locate these interfaces in the CORBA naming service, before assigning a CIWrapper to them to handle it. This is extendable to allow further client interfaces to be added, although at this stage we can't see any requirement to do so.

The Root Filter is configurable with regard to which client interfaces it talks to. You can therefore tell it not to bother talking to the database interface by commenting a line out of the configuration.

Design thoughts

The design of this class is crucial, as ultimately it could be a single bottleneck in the system. Fortunately we have a well-designed queuing mechanism, which is used throughout the server, and this allows us to regulate the data flow whilst monitoring activity.

Database Interface

Overview

The Database Interface is a "client interface", in that it is sent data by the Root Filter. It manages inserting data into a mySQL database.

Purpose

One of the reporting sides of our system is generating historical reports of a machines performance over time. This requires us to record vast amounts of data in any easily accessible format. To allow for extendibility we decided it would be best to store the data as XML in a flat table. This means any "extra" data sent by hosts in the future will be stored. The downside of this is that there is a lot of space overhead in using XML. In terms of storage an SQL database provided the most flexible solution, and mySQL seemed a good choice.

Detailed Analysis

Although relatively straightforward, there were a few catches whilst making this component. They are outlined in this section.

Receiving Data

Data is received from the Root Filter by the DBServant. Originally the servant passed data straight on for insertion into the database, but there were some caveats with doing this. All data put into the database is time stamped with a received date and time, as we can't guarantee the clocks on the hosts are all in sync. The problem was that during times of heavy database usage, such as the report generation, the inserting is delayed, often by a few

minutes. This resulted in gaps appearing in the graphs. The solution to this was to timestamp data upon receipt into the database interface.

To achieve this data would have to be queued. This then led to another problem, how we would store an XML string (we insert it straight into the data), an XMLPacket (we pull the essential[5] fields out), and a long timestamp. Tying these together required a new wrapper class which could contain all three, and thus the XMLPacketWrapper was produced.

Therefore, the XML data and timestamp are wrapped in an XMLPacketWrapper object, and this is queued for insertion.

Inserting data into the database

Inserting the data into the mySQL database is a relatively simple process. The DBInserter class grabs an XMLPacketWrapper from a queue, retrieves the required data from it, and runs an SQL INSERT command to place it into the database.

It should be noted that the database details, including username and password, are set in the configuration. It is preferable to have this in a separate file to the main configuration if the database details are considered sensitive.

The database connection is re-established with every insert, and although this isn't possibly the best solution, it does ensure that the system can more easily cope with database restarts.

When inserting data we first extract a few essential fields, such as the source machine's name and IP address, and the timestamp. These are inserted as single columns into the database, with the XML being inserted as the final row. This allows for data to be selecting on a per machine basis, or on a date basis.

Client Interface

Overview

The Client Interface is a "client interface", in that it is sent data by the Root Filter. It manages sending data to external client programs, such as Conient, or the Local Client. It provides connectivity to both TCP and CORBA clients.

Purpose

The purpose of the Client Interface is to allow externals clients to connect into the server to receive data, and to manage their connection. This involves sending the correct data to each client, and ensuring that dropped connections are fully detected and managed.

Detailed Analysis

The client interface isn't that large, but does have some complex logic in the centre. This will be explained in this section

Receiving Data

As with most components, data is received and queued by a servant class, in this case the ClientInterfaceServant. The reason for this is that we want to minimise delays over CORBA, and queuing the data for processing is the easiest way to do this.

Handling Clients

Handling of clients is done with three distinct classes for each transport. The design was originally produced for TCP clients, but was fairly logically altered to work for CORBA. Thus

[5] See the packet specification documents for details of "essential data".

the CORBA mechanism resembles the TCP one closely, with some parts maybe being unnecessary.

To start with there is a Listener class, which is primarily responsible for starting control threads for incoming clients. In the case of the TCP listener, this is a simple case of starting another thread to deal with the communication, thus allowing us to deal with more than one client at once. In the case of CORBA, it is more of a 'factory' approach, where we fire off a servant to communicate solely with a single client.

It might be worth at this point mentioning how the communication with a client works. There are, in the case of TCP connections to clients (e.g. Conient), two channels of communication. The first is the control channel, which is always open to allow the client to send commands to the server. The second channel is purely XML data, which can be stopped and started on request over the control channel. This separation makes life easier at both ends, as there's no need to attempt to sort data coming back – i.e. separate server responses to commands from the data packets. Some people have likened this setup to the FTP protocol, although the client always initiates connections. For more information about the protocol used over both these channels, please refer to the Client Connection and Client Data sections of the Protocol specification document.

To fit around this architecture, the classes that communicate with the client have been broken down into two distinct classes; the control handler and the data handler. This applies to both the TCP and CORBA classes, and in both cases the control handler is responsible for starting up the data handler upon request.

The TCP control handler receives commands from clients as simple messages, as defined in the protocol[6], and acts accordingly. These commands are listed here:

STARTDATA – starts the data channel.
STOPDATA – stops the data channel.
STARTCONFIG – allows the client to get configuration from the central configuration system.
SETHOSTLIST – allows the client to set a list of hosts it wishes to receive data about.
DISCONNECT – disconnects the client from the control handler

The first command, STARTDATA, asks the control handler to initiate sending of data. This is done by creating a data handler class with the appropriate details. The data handler has an internal queue, and a reference to this queue is passed to the packet sorter (which will be explained later). The data handler then pulls data from it's queue and keeps sending it down to the client, over the data channel. This continues until such a point as STOPDATA is sent, which indicates to the control handler that the data should no longer be sent. To action this, the control handler sends a shutdown request to the data handler. The data handler deals with closing itself down and disconnecting the link, and will then be left for garbage collection.

The STARTCONFIG command allows a connected client to retrieve configuration from the central server configuration. An example of such configuration would be getting details of host update times to display visual timing of expected data arrival. This is a fairly trivial process of the client asking for a property, and the server returning the value, until such a point as STOPCONFIG is sent. Note that there are restrictions in place to ensure the client is only getting configuration it is permitted to view.

The least complex of commands is SETHOSTLIST. This simply registers a preference of what hosts the client would like to receive data about. This command can only be used when the client is not receiving data. When STARTDATA is used the host list is sent along with it, so no change will happen until STARTDATA is sent. By default the host list is an empty string, "", which indicates no preference – i.e. all hosts. The PacketSorter uses the host list.

The DISCONNECT command is fairly straightforward, although it does require correct closing down of the TCP connections, deregistering, and shutting down data handlers.

[6] See Protocol specification for further details

All of the above commands are clearly visible in the TCP control handler's main loop. The data handler isn't complex, it simply reads data from a queue and sends it over a TCP link.

The CORBA handlers are a pretty similar affair, with the same overall structure, and same commands. The main difference is, of course, that the string commands sent over TCP are replaced by method calls. There are, however, other subtle differences in how the data is sent around, mainly due to the CORBA structure.

The control channel works by having a servant at the server end that the client can make calls on; this is similar to the TCP arrangement. The data channel works differently, with the servant being started on the client end, and the server making calls on it. To do this the client passes a reference to it's own servant upon connection, and this is then given to the CORBA data handler to send data back to.

Another complexity arises with disconnecting, as the appropriate servants must be released from the POA. This happens in the control handler's disconnect method, and merely requires the servant's CORBA OID to be given to the POA deactivate_object method. Without this extra behaviour a build up of control handler's would occur as CORBA clients connect and disconnect, shadow servants would build up and not be garbage collected.

Distribution and Sorting of data

nb. this section requires an understanding of how LinkedLists and HashMaps work.

So far we have looked at how data is passed into the client interface, and how it's taken out at the other side, but that leaves the chunk in the middle. This section of the client interface sorts data, sending it to the clients that wish to hear about it, and has consequently been named the PacketSorter.

There are two sections to this class, the registration and deregistration of handlers, and the sorting and distribution of packets. Both are tightly linked, and thus synchronisation issues have been a major concern here.

The registration part is handled by passing a reference to a queue and a requested host list to a method on the PacketSorter. The queue is the one in the data handler that's being registered, and the host list is that which the client has requested (or an empty string indicating all hosts). The PacketSorter maintains a series of LinkedLists, all contained in a HashMap, using their hostname as a key. Each LinkedList contains a series of queue references belonging to those clients interested in the given host. There is also a separate LinkedList that contains queue references belonging to those clients interested in all hosts. Although this seems quite complex, it isn't too hard to maintain.

The de-registration process is the exact opposite of the above process. The queue references from the disconnecting client are removed from all the LinkedLists, based on the host list given to the method.

With this structure in place distributing the data is a fairly simple process. As each packet of data arrives the source host is looked up, and using this name the relevant list is pulled from the HashMap, and the packet placed in every queue reference found in that list. Finally the packet is sent to all queue references found in the all hosts list. Some extra logic is also added to send any non-data and non-heartbeat packets, such as queue information, to every client.

This process is fairly logical, but has a high degree of coupling and could cause potential bottlenecks. Simplification of this section would be great, but the fact remains that there are lots of hosts sending data to lots of clients, which some awful sorting in the middle.

Local Client

Overview

The local client acts as a client to the server, and handles monitoring the data and generating alerts as required. However, as there would only feasibly be one of these in the system, it is run as a component of the server, and is therefore named the Local Client.

Purpose

The original design plan was to have a simple mechanism that generated an alert by e-mail when something went wrong. However, when the implementation began it soon became apparent the problem was a lot more involved and complex.

The overall aim was to have a mechanism that generated alerts, with varying degrees of intensity, through differing delivery channels. It also needed to be able to regulate alerts so that not too many were sent for the same event (or sending more if the monitored event escalates), yet keep an event open by sending repeated events over time.

The end design allows monitoring of a multitude of different host data items, and alerting through a variety of mechanisms. This is all tied together in an extendable and pluggable architecture, which fits with the ethos of the system.

Detailed Analysis

The Local Client is broken down into 2 distinct sections, the Monitor's and the Alerter's. The Monitor's are responsible for monitoring data and deciding when an alert should be raised, whilst the alerters deal with delivering an alert. The odd section is the WebFeeder, which delivers alerts and data to the web reporting system.

It should be noted that the Local Client has a multitude of Queue objects, which gives lots of scope for data to build up. Fortunately there is a monitor specifically watching queues, and any problems will cause an alert to be raised.

Another issue is the resource usage of the Local Client. It is very intensive, relatively speaking, as it does a lot of processing and stores a lot of residual information. On our test setup of between fifteen and twenty hosts, the Local Client consumed approximately the same resources as the rest of the server put together. We have therefore found it advisable to run it in a separate JVM to the rest of the server. One of the advantages of the LocalClient over other components is the system is that it plugs into the ClientInterface in the same way that TCP clients do and thus it can be stopped started independently of the rest of the system and the ClientInterface will handle deregistration.

Monitor Architecture

The monitor's are all very similar, in that they take host data packets, analyse them, and then decide whether to generate an Alert. They are all constructed by the MonitorManager, and all extend the MonitorSkeleton abstract class. The MonitorManager receives all XML packets delivered from the Client Interface part of the server. It then sorts these into relevant queues to make life easier for the Monitor's.

Each Monitor hooks to one of the queues in the MonitorManager, and processes each packet it receives. There are 4 queues available "data", "heatbeat", "other" and "all". The separation of these queues attempts to reduce the load on Monitors by only having them check packets they know they will be interested in. Typically each monitor will only be interested in a single specific subset of data from a packet. Each monitor is independent of the others, and can operate at it's own speed.

Internally an individual Monitor first analyses, in the analysePacket method, a packet it receives. This usually involves pulling out the item(s) of data it is interested in, and passing it to the checkAttributeThreshold method. This method returns the threshold, if any, which the

data has passed. Both of these methods are specific to each monitor, rather than being in the skeleton class.

The next step, which is common across all monitors, is to process the value and threshold, in the processAlert method, to see if an alert should be raised. This takes into account previous alerts, and can escalate an alert to a new level if required. An individual alert starts at a low level, such as NOTICE, and if the problem persists will ultimately be escalated to CRITICAL. Finally, if an alert needs is raised it is passed to fireAlert for generation and sending.

All monitors follow this last stage, and all of them drop alerts onto a special queue. These alerts are wrapped in the form of an Alert object. This queue then feeds into the next section, the Alerters.

A point to note is that throughout the Monitors a Register class is used to maintain the alert level, previous alerts, and gain configuration for the monitor. This class is very tightly woven into the Monitor architecture.

The Monitor part of the Local Client, and indeed the whole server, is one of the most challenging logic wise to write and understand. A lot of data is used, stored and processed, and consequently there is plenty of room for error.

For more information about the types of monitors available and how to configure them appropriately, please refer to the server user guide.

Alerter Architecture

The Alerter architecture is similar to the Monitor setup, but fortunately much easier to follow. Data is fed in through a single queue, which is fed in from the Monitors (see last section). This contains Alert packets which must be delivered.

Each Alerter extends the AlerterSkeleton class, and is created by the AlerterManager. Once active they pull data from the incoming queue and examine the Alert object. They have a level at which they deliver alerts, and if (and only if) the incoming Alert is equal or past this level do they process it. This allows each Alerter to run at different levels, which might be useful if you only want the highest alerts by e-mail.

The processing part of an Alert generally involves making a textual message from the Alert data. This is usually done by reading template messages from the configuration, and replacing key words with actual values. The message is then delivered by whatever mechanism the Alerter serves – examples are E-Mail and IRC.

This structure is again designed to be easily extendable to allow further Alerter's to be easily written and plugged in.

Again, for more information about the types of alerters available and how to configure them appropriately, please refer to the server user guide.

WebFeeder

The WebFeeder is the last part of the Local Client, and is relatively simple in comparison to the rest. Its job is to feed all data packets and all alerts to a file structure somewhere on a disk. This data is then read in by webpages to generate a web front end for viewing latest data and alerts. The web interface is discussed in another document.

The interesting part of this section is how it acquires all the data packets and all the alerts. Two small "servant" classes were written to act as a Monitor and an Alerter. They sit in the chain with their siblings, and are duly fed all the required data. The main WebFeeder is a singleton class, and thus they can both get a hook on it and deliver all the data to it.

Once the WebFeeder has the data, it just needs to write it out to disk. The paths and filenames are read from the configuration, and the all the data is dumped out using appropriately written toString methods on the incoming objects.

Although this may seem straightforward, there are a few caveats. The first is that once alerts return to an OK level, or reach a FINAL level, they must be cleaned up. This requires the WebFeeder to cycle round clearing out all the relevant alerts. This is done by making the WebFeeder a thread, and having a configurable delay between cycles. When an Alert is cleared out we also clear out the directory left behind if no more alerts remain.

There is also a more specialised case, the Heartbeat FINAL alert. When this alert arrives a host has been down for some time, and the system (using the configuration) has decided that it is not coming back. This also means we need to clear out old and stale alerts, because they will never be set to an OK status. This logic is built into the afore mentioned cycle.

It is important to note the use of filenames the data is written out with. In the case of alerts, we write them out with the date they were raised, as it is unlikely we will get two for the same host at the same time. Each escalation overwrites the previous, meaning we only see the latest Alert of each raised alert. We also make special cases of OK and FINAL alerts by attaching a .OK or .FINAL extension to them. This allows the checking loop to spot them without having to parse the data. There is an even more specialised case for Heartbeat FINAL alerts, which have the extension .HB.FINAL.

The data packets, however, are written out with no complexity at all. Each host only has the latest data packet, and they are never cleaned up. It is left up to the webpages to indicate the age of the data.

As a final note, the WebFeeder clears out all alerts on starting up, as no alert can be carried from one incarnation of the Local Client to the next. Data packets are always left, and it is up to the user to delete old directories manually.

The WebFeeder is of course fully configurable, and can be turned off if required. For information regarding this configuration, please refer to the server user guide. For information about how the web interface operates and how it should be used, please refer to its maintenance and user guide.

Util Package

The util package is a collection of useful classes used in the server. These not only reside in a separate package, but are actually built into an independent JAR file. This allows other parts of the i-scream system to make use of them. One semi-formal rule of the util package is that the classes must not depend on anything else in the server, such that they can be used by external programs.

DateUtils

Description

The DateUtils class provides a variety of methods for working with dates. These include methods to return the time at the start of today, and to turn a long time period into a presentable String. These are methods that generally come in useful, but are not provided in the Java API.

FormatName

Description

This FormatName class is used by most classes to generate a string representation of their name. This allows lines in the logfile to have a common naming format, without duplication of code. The getName method provides this functionality by taking various details about a class and returning a formatted string. There is also a method in this class to format a line for the logfile, again centralising this behaviour.

Queue

Description

The Queue class has been widely used throughout the system, mainly for storing XML. It has not only provided queuing facilities, it has also given a unique way of linking a single thread to one or more other threads in an independent manner.

Definition of Terms

Various terms will be used to discuss the Queue and it's features. They are defined here.

- Queue
  The single instance of the Queue class being used.
- Producer
  Usually the single object generating information, although there could, in theory, be more than one.
- Consumer(s)
  An object requiring data from the Producer.
- queue
  A queue within the Queue class.

Features

The queue provides an extensive range of features, mostly geared towards the needs of this system, although it could easily be applied to any threaded environment needing a queuing mechanism. Here is a full list of features

- Support for a multi-threaded environment

  In a system where the producer and consumer threads are separate, it can be hard to coordinate the adding and removal of data from a queue. It may also not be beneficial for the consumer to keep attempting to get data until some is available.

  To solve this problem the Queue provides a get() method that allows blocking if no data is available, thus halting the consumer thread.

- Support for multiple consumers

  This does away with the need for creating multiple instances of the Queue and populating them all with the same data.

  Internally this is done by multiple queues which are populated using a single add() method. From the perspective on the producer, this makes things much more straightforward. Each queue is independent, and therefore allows consumers to operate at different speeds.

- Support for dynamic creation/removal of queues

  This allows a consumer to request removal of a queue it may be using. This helps to keep things tidy if a consumer needs to be shut down - i.e. the internal queue will no longer be populated, and any remaining data will be left for garbage collection.

  A queue will be automatically created upon calling the getQueue() method, which again makes life easier for a system where consumers may be coming and going.

- Built in monitoring of the internal queues using XML

  Using a queueMonitor the internal queues of a Queue can be monitored externally. This is done by generating XML statistical data which can be placed into the queue for processing. This is very specific to the i-scream environment, and allows us to keep track of data flow within the system.

- Queue size limiting with choice of removal algorithms

  An upper limit can be placed on the internal queues, preventing problems and slow consumers from causing memory overflows. There are a choice of algorithms to be applied when a queue is full and new data arrives.

Usage

The Queue, although fairly complex, is quite easy to use. The basic features will be explained here, but more detail can be found by looking at the code and javadoc.

Constructing a simple Queue.

```
Queue q = new Queue();
```

Adding objects to the Queue.

```
q.add(o);
```

Getting a queue setup.

```
qID = q.getQueue();
```

Getting an object from your queue.

```
Object o = q.get(qID);
```

Discarding your queue when finished with.

```
q.removeQueue(qID);
```

It is extremely important to do this last stage if the queue has been finished with. If this is not done data will continue to be added to the queue until the system runs out of memory. Once a consumer has been assigned a qID it is responsible for using it, and ensuring data is removed.

Smtp

Description

The Smtp class was taken from the GJT[7] and scaled down to suit the requirements of this project. It provides a simple method of sending an e-mail to an SMTP server from another class.

Usage

Typical usage is as follows, for a basic e-mail to one person.

```
Smtp smtp = new Smtp("smtp.mydomain.com");
smtp.setSender("me@mydomain.com");
smtp.setTo("you@yourdomain.com");
PrintWriter out = smtp.getOutputStream();
out.println("Subject: Test Message");
out.println("This is a test message, first line");
out.println("Yet another line of test message");
out.println("Thanks, me.");
smtp.sendMessage();
smtp.close();
```

This sends an e-mail to “you@your.domain.com”, from “me@mydomain.com”, with the subject “Test Message”. The body is the three lines following the subject.

StringUtils

Description

There are two useful methods provided in the StringUtils class. These are methods we needed to use in more than one place, and found Java could not provide.

More details can be found in the javadoc.

XMLPacket and related classes

Description

The XML classes found in the util package allow XML strings to be parsed into a flexible XMLPacket object. The XMLPacket object allows you to easily get at any item from the original XML data using a single method call.

[7] Giant Java Tree – http://www.gjt.org

The parsing requires the external JAXP[8] libraries, although these are included in the server distribution. XML is checked for conformance to XML standards in the parsing stage, and any bad XML is rejected with an InvalidXMLException.

These classes are used throughout the i-scream server, and other components. They were written to utilise the existing libraries and to take the output from these libraries and turn it into a format we could make use of – the XMLPacket.

Usage

Using the XML classes is pretty simple. Here is a basic example to parse a simple XML string into an XMLPacket object.

```
String xml = "<packet><tag>data</tag></packet>";
XMLPacketMaker xmlPacketMaker = new XMLPacketMaker();
XMLPacket packet = xmlPacketMaker.createXMLPacket(xml);
```

Note that exception handling is not shown here. It should also be noted that a single XMLPacketMaker could be used more than once, so it is not necessary to recreate it for every string to be parsed.

The resulting data from the XMLPacket could be retrieved as follows.

```
String xmlData = packet.getParam("packet.tag");
```

The xmlData string would then contain the value “data”.

[8] http://java.sun.com/xml/

Plugin Architecture

One of the things that has been set out since the start is the ability to have a plugin architecture. By this we mean it is easy for someone to write a new plugin, conforming to a specification, and easily slot this into the system – without changing any other code.

Unfortunately this isn't a dynamic architecture, so it won't automatically spot new plugins and use them, they will have to be activated in the configuration at system start only.

There are two components that make use of plugins; the Filter contains Filter Plugins and Service Check Plugins, and the Local Client that contains Monitor Plugins and Alerter Plugins. These will be looked at it more detail here.

A similar design is followed with the logging system in the Core, but this was written before plugins were introduced, and as such does conform to the same setup.

The actual plugins themselves will be explained in more detail in the user documentation, and the javadoc/code will detail the actual implementations. This section merely looks at to concepts and types of plugins.

Plugin Managers

In all four types of plugins there exists a related plugin manager. It is generally the job of the manager to construct the plugins listed in the configuration, and as reflection is usually used this needs to be handled in one place.

The managers are also responsible for activating the plugins, and in some cases passing data to them. It is up to the manager to keep tabs on the plugins, and if implemented unload and reload them.

Filter Plugins

A filter plugin is designed to examine a single piece of XML data, in an XMLPacket, and decide whether it should be rejected or not. This is currently a fairly simple task, although it is envisaged the complex plugins could be written to watch for packet storms.

These plugins conform to the PluginFilter interface, which defines that they take an XMLPacket and return a boolean response. This response indicates whether they allow the packet to pass or not.

Filter plugins can be identified by their name - <name>__Plugin

More details of how this fits in can be found in the Filter section of this document.

Service Check Plugins

A service check plugin also resides in the Filter, but lives in the TCP heartbeat side of things. It's task is to run a check on a specific service, such as HTTP. It does this by making a connection to a known port, checking to see if the response matches what is expected. It then returns XML data to represent this.

These plugins are chained together into a pipeline to be run on hosts when they send a heartbeat. The exact set of plugins to be run depends entirely on the configuration, and the manager class handles this. The manager collates the responses from multiple service check plugins, and sends this along in a heartbeat packet.

All service check plugins must extend the ServiceCheckSkeleton class, which provides some basic functionality. In doing this they implement the PluginServiceCheck interface, which is what is expected.

Service check plugins can be identified by their name - <name>__ServiceCheck

Again, detail of how this fits in can be found in the Filter section of this document.

Monitor Plugins

The Monitor plugins can be found in the Local Client, and are responsible for analysing a specific type of data – such as CPU loads. A Monitor plugin can generate an alert if the data it is monitoring passes a certain threshold.

These plugins together form the list of all the data that will be monitored for alerts. They are complex to write, but this is expected from the tasks they perform.

All Monitor plugins should extend the MonitorSkeleton class, which provides them with basic functionality. Ultimately they will be expected to implement the PluginMonitor interface, through extending the MonitorSkeleton or otherwise.

Monitor plugins can be identified by their name - <name>__Monitor

This architecture is discussed in depth in the Local Client section.

Alerter Plugins

Alerter plugins are also found in the Local Client, and perform the task of delivering Alerts through some mechanism. An example of such would be E-Mail. They are relatively simple, and merely have to know how to do deal with an Alert object.

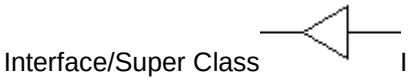
They must extend the AlerterSkeleton, which in turn means they implement the required PluginAlerter interface. They are managed by the AlerterManager.

Filter plugins can be identified by their name - <name>__Alerter

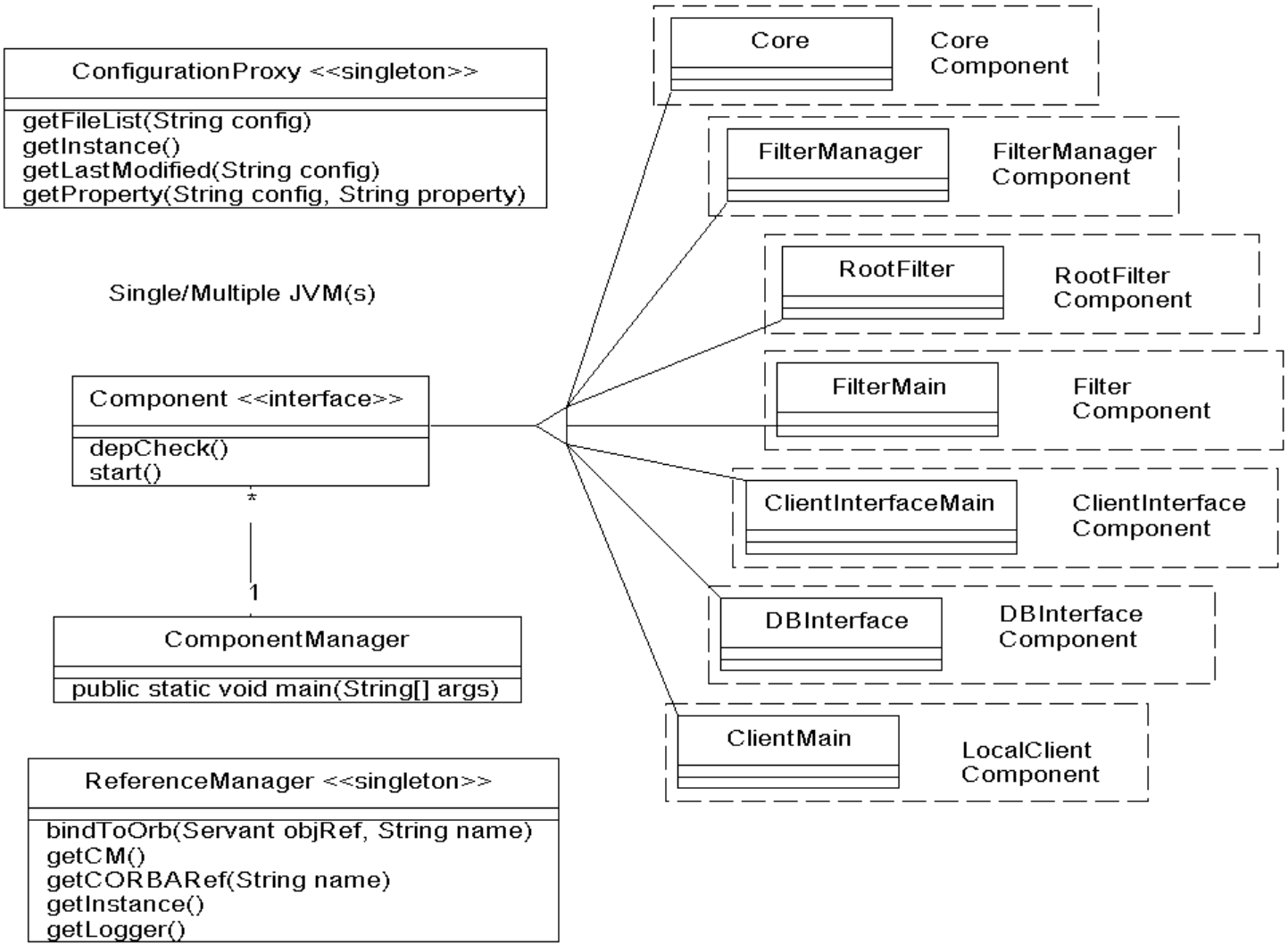
More extensive detail can be found in the Local Client section of this document.

System Diagrams

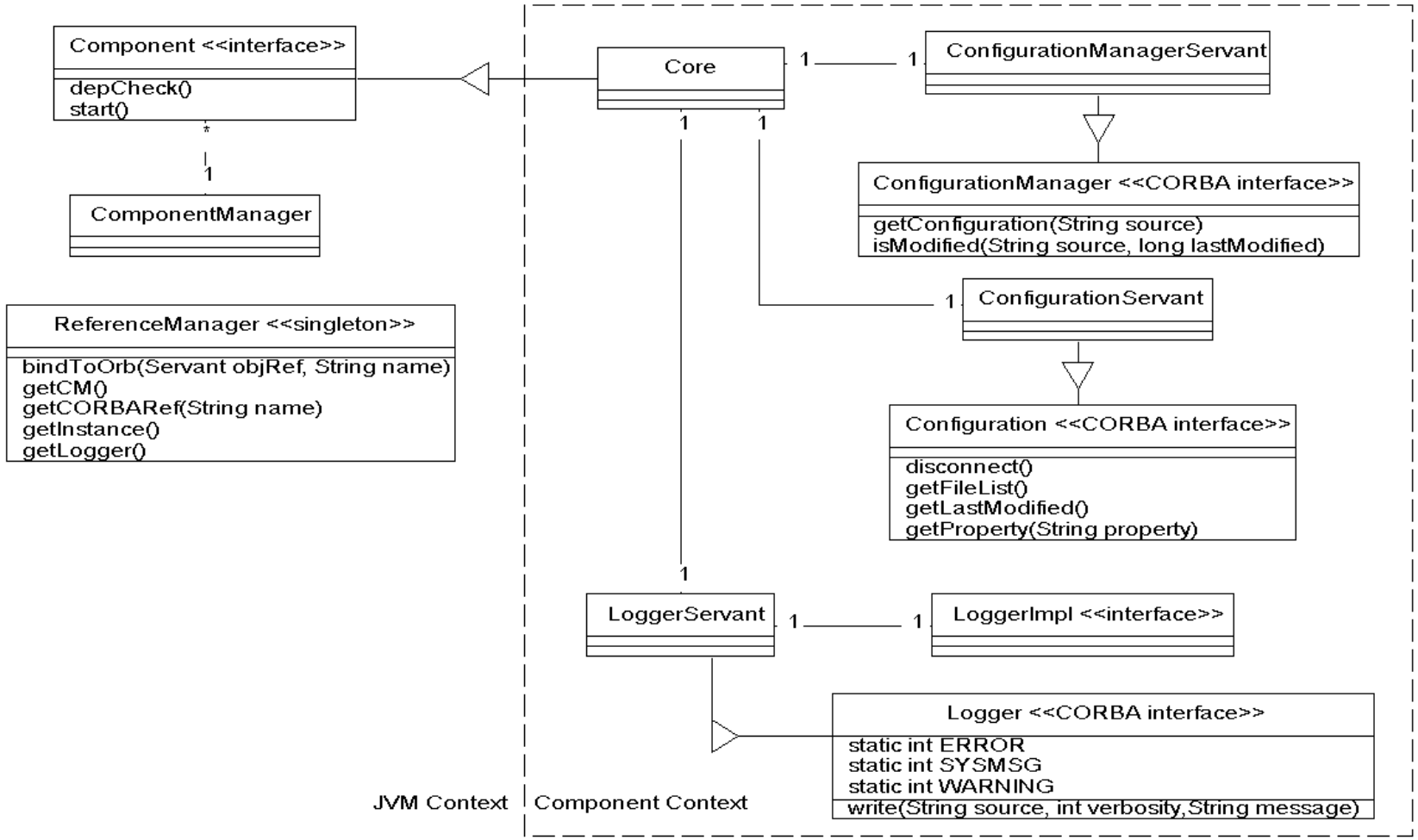
This diagram is intended to show the connections between the classes and the main methods available within the i-scream server system. Although it is based on a UML class diagram it has been modified to act as a simple visual aid to describe how the various parts of the system relate to each other. Some standard indicators do remain, such as the links between classes, e.g. 1 ----- *.
There are also some added non-standard features. The arrow heads on the links between some classes (such as the Queue class), are intended to show the direction of data flow and the following symbol is used as an indicator of “Extends” and “Implements” relationships between classes.



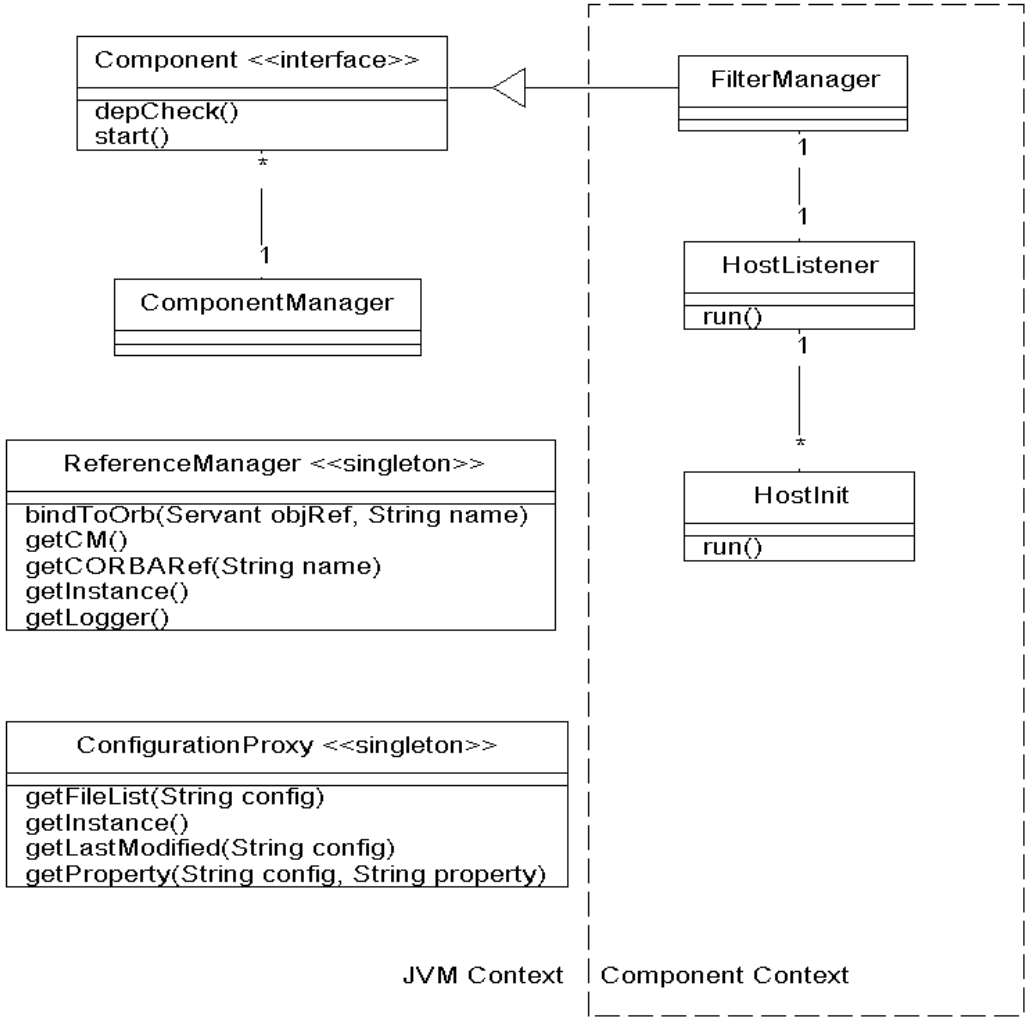
Overall server component architecture



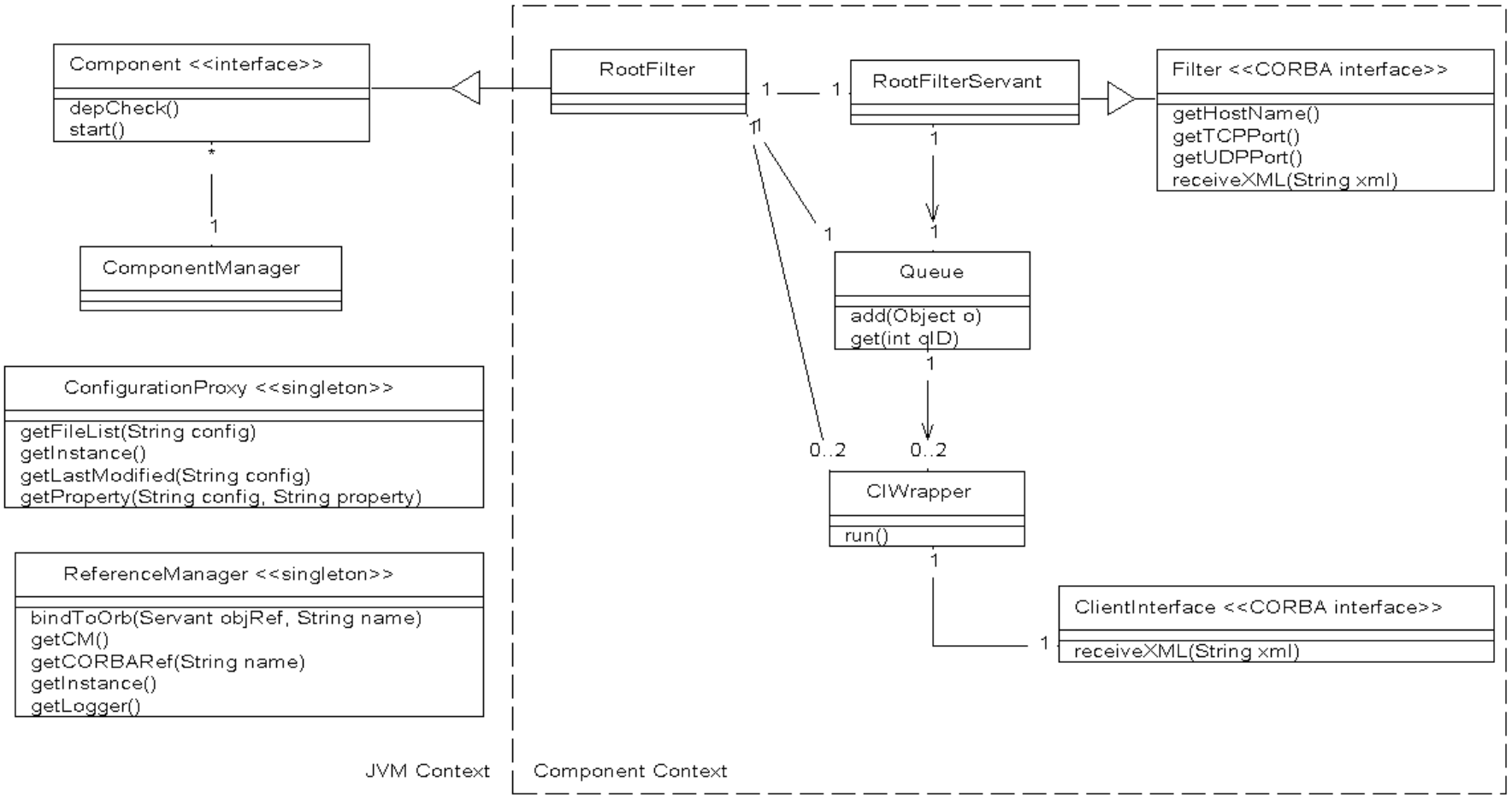
Core Component



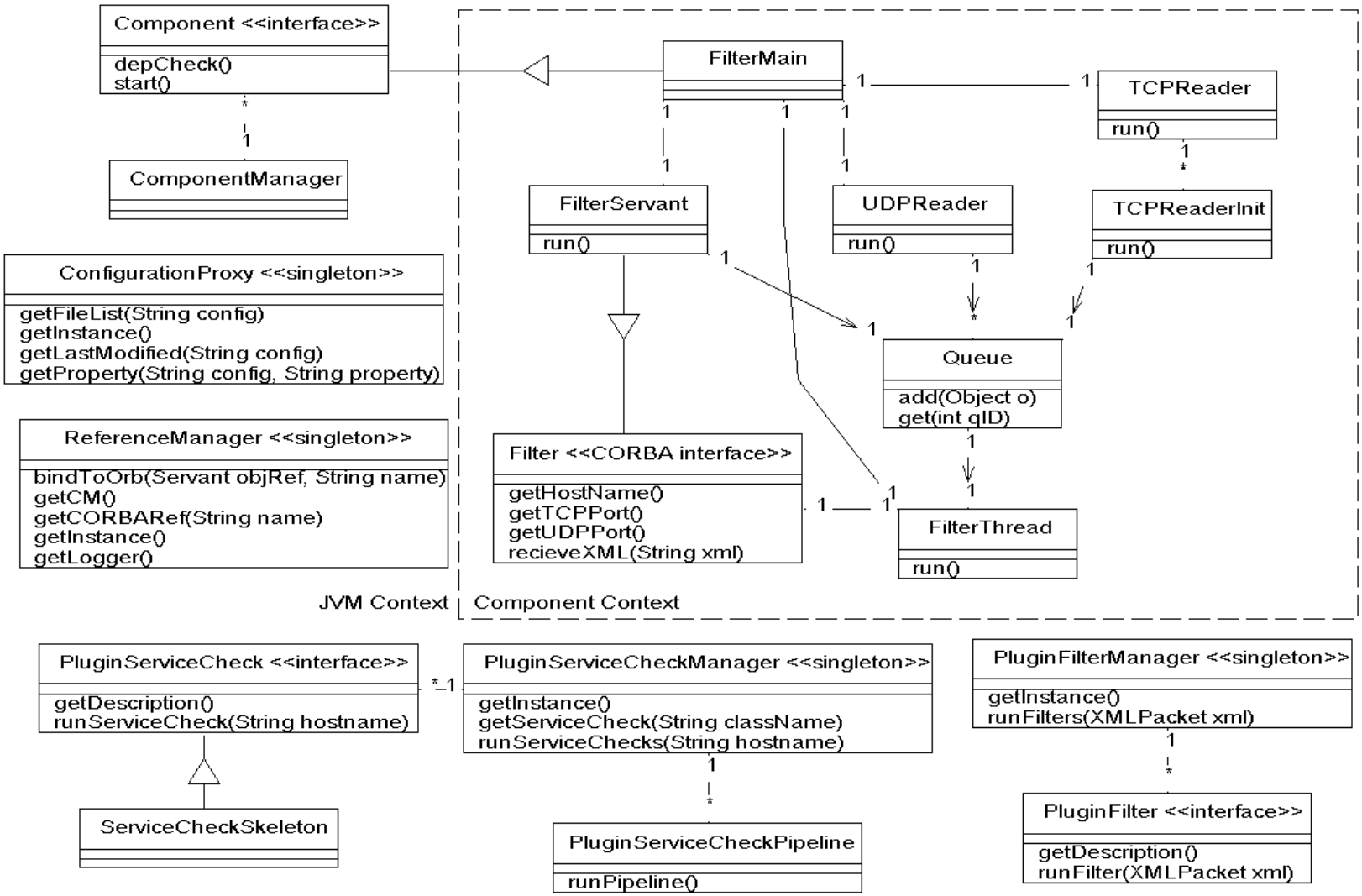
FilterManager Component



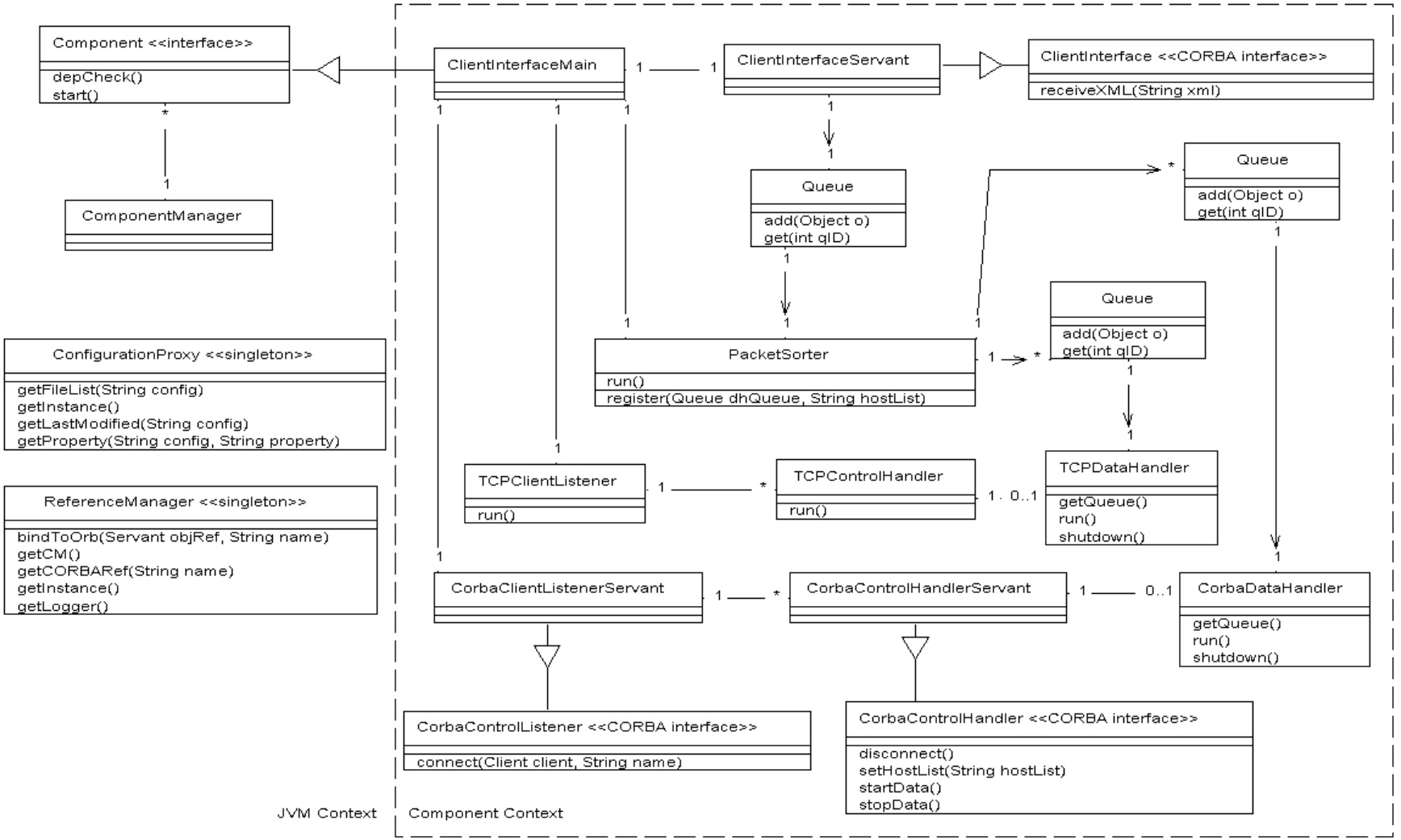
RootFilter Component



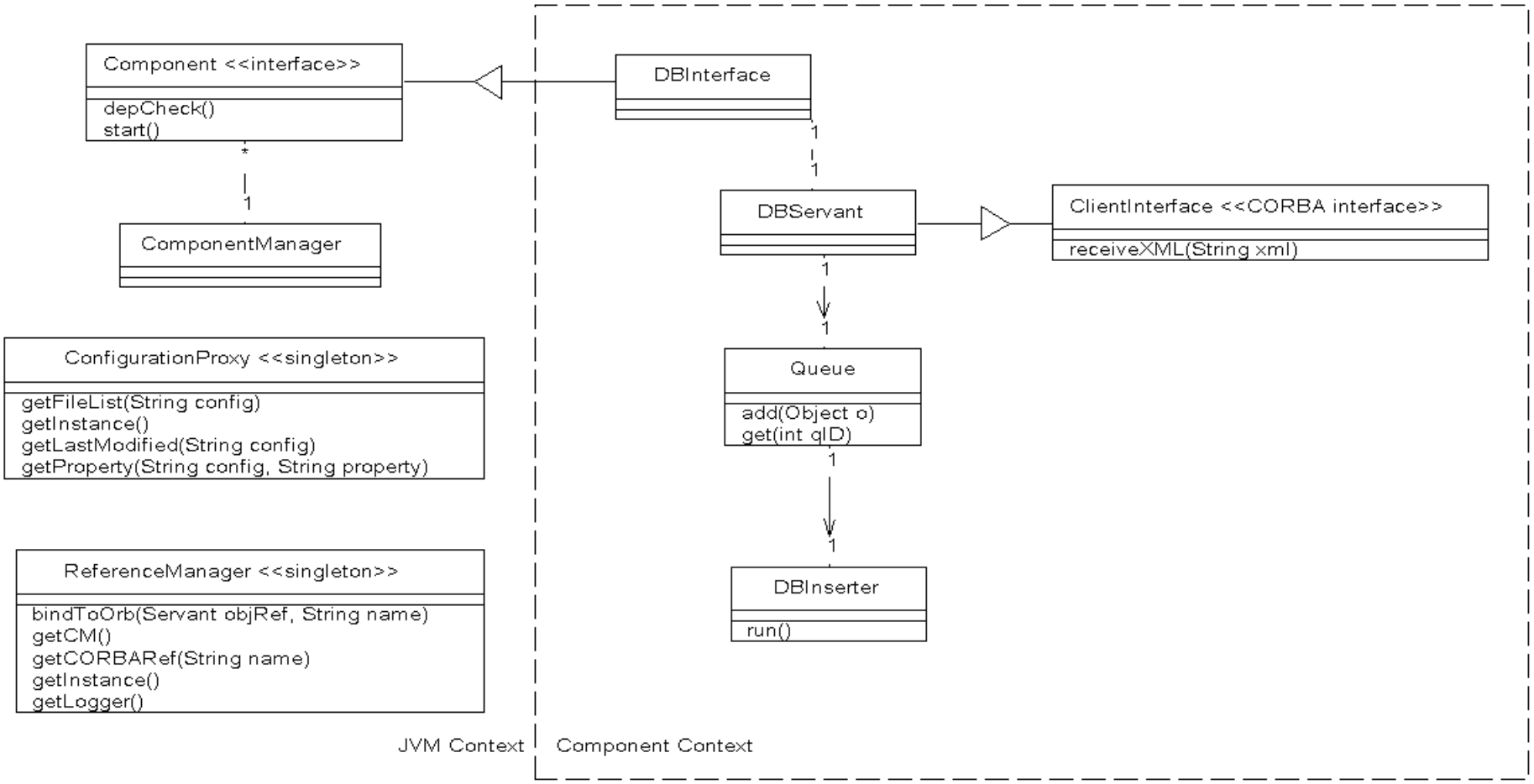
Filter Component



ClientInterface Component



DBInterface Component



LocalClient Component