



**Developers Guide to ORBs and Remote
Method Invocation**

Contents

1	Preface	1
2	ORB features.....	2
2.1	Backward compatibility	2
2.2	Fault tolerance	2
2.3	Compression	2
2.4	UTC.....	3
2.5	Unicode	3
2.6	Error propagation.....	3
3	Client ORB	3
3.1	Connection.....	4
3.2	Automatic reconnection	4
4	Server ORB	4
4.1	Multithreaded server	6
4.2	Server connection limit	6
4.3	Thread pools	6
4.4	Server Shutdown.....	7
4.5	Server Kill	7
4.6	Server Status	7
4.7	Terminate events.....	7
4.8	ServerORB callbacks	8
4.9	Log	8

1 Preface

ADK 2.0.0 includes an enhanced remote method invocation mechanism. The improvements affect things as basic as the transmission protocol up to the server and client classes and include the following:

- Support for Unicode characters
- Better fault tolerance
- Support for compression of messages
- Support for UTC time/date format
- Error/exception propagation from server to client
- Automatic reconnection
- Multithreaded Java services
- Improved service control and information

The protocol from ADK 1.3.6 and previous versions will be referred to as the old protocol, and the protocol introduced with ADK 1.3.7 and ADK 2.0.0 will be called the new protocol.

This document will explain the new functionality and how to take advantage of it.

2 ORB features

2.1 Backward compatibility

By default, the client ORBs (`InternalORB` and `InternalSessionORB`) run with the protocol from ADK 1.3.6. The default can be changed using the property:

```
agotor.orb.request.oldformat
```

However, the behavior can be set specifically for every invocation using the `InternalORB`. The server will, however, always answer a question using the same format as the client, i.e., a client sending a request in the old format will receive an answer in the old format no matter what the format setting is on the server.



If you use the old protocol then the compression, error propagation, UTC, and Unicode features are not available. Furthermore, any future development is centered on the new protocol and new features are not likely to be supported when the ORB runs with the old protocol.

2.2 Fault tolerance

Each request is marked with an ID and only responses with the same ID are accepted. This helps timeout problems. In fact, if a server is very slow, a client could receive a response after the request had timed out. In this case, since the timed out response has an old ID, it will be discarded. In this way, the protocol is more robust against time out errors.



The ID filter also increases the security by accepting only the right responses.

The fault tolerance is increased also in the form that the protocol is able to clean the transmission channel for incompletely read data. This could happen, for example, after a crash or due to an IDL mismatch between the server and the client.

2.3 Compression

The communication between the server and the client can be compressed in order to reduce the amount of data send through the net and increase the speed. The feature can be activated by the property:

```
agotor.orb.protocol.compression
```

Currently the only possible value is `gzip`. The feature can also be activated or deactivated directly from code using the `InternalORB` class.

As compressing data will consume some processing time, compression is best suited for large messages containing plain text.

2.4 UTC

UTC means Universal Time Coordinate. This is a standard for expressing a date/time independently of the time zone where the application is running. The old protocol sends and receives the time relative to the local time zone. Therefore, if you have two machines running in two different time zones then the time transferred between the two machines cannot represent the same time. The new protocol transfers the time represented in the UTC format, which means that the time value is the same.



This does not mean that its display/string representation will be the same, because it is localized to your machine; but the important thing is that the two values represent the same time.

2.5 Unicode

The transmission protocol can handle Unicode strings and characters, which means that you can transmit strings in languages like Chinese, Russian, and Arabic. If Unicode strings are used then they are encoded using the Java system default encoding or a specified encoding. The encoding can be defined using the property:

```
agotor.orb.protocol.encoding
```

If you wish to transmit Unicode in another encoding than the system default, set this value to UTF8, UTF-16, or another of the formats supported by Java.

See <http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.HTML> for more information.



If you run with the old format (see above) then only ASCII strings and characters can be sent. This means, that if you are dealing with non-ASCII characters then they will not be transmitted correctly.

2.6 Error propagation

If a server request raises an error (e.g., a Java exception), then this is forwarded to the client in the form of the RemoteServerException.

The exception contains information like an error kind (in Java, this is the Exception class) the error message, and the error position (in Java, this is the stack trace).

3 Client ORB

The client ORBs are still the `InternalORB` or `InternalSessionORB` used previously. They have, however, been added some extra functionality in ADK 1.3.7.

The automatic login in the `InternalSessionORB` is *no longer available* for security reasons. This means, that if a user forgets to logout, then the login session will time out and will not be available any more. The user will need to login again.

If the `InternalSessionORB` should remember the user login information and should make an automatic login when a session has expired, then other people could access the user data if the user forgets to logout and shut down the application (e.g. just leave the machine with the web browser open on that page).

3.1 Connection

When a client tries to establish a connection then the InternalORB automatically repeats the attempt a certain amount of times before raising a ConnectionException. The number of attempts and the time between each attempt are defined by two properties:

```
agotor.comm.socket.attempts  
agotor.comm.socket.delay
```

The delay is measured in seconds.

3.2 Automatic reconnection

The InternalORB is capable of automatically reconnecting to the server if the connection should be lost. This could happen because the server or the broker have been stopped and then restarted. If the reconnection is not possible then the ORB raises a ConnectionBrokenException.

4 Server ORB

ServerORB is a new ORB class, which replaces the WebAnswerORB class. WebAnswerORB is still available but the class does not support a lot of the new functionality like multithreading, or the new server functions like shutdown and status. The use of ServerORB is, however, very similar to WebAnswerORB:

```
public class MyServer implements MyServerInterface {  
    private class MyCallback extends ServerORBEvents {  
        public void beforeShutdown() {  
            System.out.println("Cleaning before the server is shut down ");  
        }  
  
        public void afterShutdown() {  
            System.out.println("Cleaning after the server is shut down");  
        }  
  
        public void aboutToKill(long sec) {  
            System.out.println("abort all the requests which take longer time than " +  
                sec + " sec");  
        }  
  
        public void onKill() {  
            System.out.println("abort all");  
        }  
  
        public void status(InfoList info) {  
            info.addValue("MyStatus", "OK");  
        }  
  
        public void statistics(InfoList info) {  
            info.addValue("Called", String.valueOf(calledTimes));  
        }  
  
        public void info(InfoList info) {
```

```

        info.addValue("MyServerVesion","1.0.0");
    }

}

public static void main(String args[]) {
    MyServer myserver = new MyServer();

    ServerORB serverORB = null;

    try {
        serverORB = ServerORB.makeServerORB(LOGGROUP, args);
        serverORB.setCallback(myserver.new MyCallback());
        serverORB.bind(myserver);
        serverORB.run();
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}

public MyServer() {
}

// Service implementation ....
}

```

The command line arguments for ServerORB are:

Argument	Description	Default (agetor properties)	Default
-p<num> or --port:<num>	The IP port number		0
--maxconn:<num>	The max number or connections	agetor.orb.server.connections.max	10
--maxreq:<num>	The max number of concurrent requests	agetor.orb.server.requests.max	1
--minconn:<num>	The minimal number of thread in the connection thread pool	agetor.orb.server.connections.startpool	1
--minreq:<num>	The minimal number of thread in the request thread pool	agetor.orb.server.requests.startpool	1

The above commands will be explained in the following.

4.1 Multithreaded server

The `ServerORB` is used as the communication server in a service. The server is capable of running many concurrent requests. Each request runs in its own thread.

However, the server is capable of limiting the number of concurrent requests. Setting the maximum number of concurrent requests does this. A default maximum is specified by the property:

```
ageton.orb.server.requests.max
```

A service can specify its own maximum by the command line parameter `-m<number>` or by passing the value in the `ServerORB` constructor.

A maximum is nice if you are dealing with limited resources. If you set the max equal to the amount of available resources then the `ServerORB` can manage the concurrent request in such a way that any extra requests will wait for a free resource.

4.2 Server connection limit

It is possible to limit how many connections the server allows simultaneously. Any extra connection will be refused. The default limit is again set by a property:

```
ageton.orb.server.connections.max
```

You can customize this by the command line argument `-c<number>` or in the `ServerORB` constructor.

4.3 Thread pools

Each connection and each request runs in its own thread in the `ServerORB`. In order to increase the performance the server pools these threads. It is possible to set the minimum number of threads in the pool. If all the threads are destroyed every time they are not needed the server will have a longer response time. Having a minimum amount of threads will shorter the response time.



The server start up time will be increased slightly but this is not so important because it is only when the server is started.

The default minimum number of threads is specified by properties but they can be customized via command line parameters or the `ServerORB` constructor.

	Property	Command line
Connection thread pool	<code>ageton.orb.server.connections.startpool</code>	-- <code>minconn:<number></code>
Request thread pool	<code>ageton.orb.server.requests.startpool</code>	-- <code>minreq:<number></code>

4.4 Server Shutdown

The `ServerORB` is capable of stopping the server in a controlled way. The shutdown procedure is:

1. Set the shutdown flag
2. Close the server for any new connections. The clients will receive a timeout exception (this will be changed in a future release).
3. Remove any waiting requests. The clients will receive a timeout exception (this will be changed in a future release)
4. Await that the running requests terminate
5. Close the connections



The connections are closed very late in the process. This means that it is still possible to manage the server while it is shutting down. `InternalORB` has a shutdown method that can shut down the remote service. You cannot invoke the shutdown via the broker. You need to connect directly to the server (host and port).

4.5 Server Kill

The kill task is similar the shutdown task except that the kill will wait only for a certain amount of time for the requests to terminate. Thereafter the request threads are stopped.

You can call a kill command after a shutdown command. In this case any wait from the shutdown process will be aborted.

Again, the `InternalORB` has a shutdown method that can shut down the remote service. You cannot invoke the shutdown via the broker; you need to connect directly to the server (host and port).

4.6 Server Status

A ping command can only tell about the connection between a client and a server. The `ServerORB` can also return status information. The status is a string that contains the server status plus some custom service information. The string can be plain text, XML or HTML. The client asks the server for the status in a certain format (plain text, XML, HTML). However, the server can return the status in a format different than the requested.

The status information is composed of the server status, the connection thread pool status, the request thread pool status and the custom status. The status from the to thread pools can be activated by setting the below properties true:

```
agotor.socket.status.connpool
agotor.orb.server.status.reqpool
```

The custom status is read by using the callback mechanism. See the callback section for more information.

4.7 Terminate events

The `ServerORB` will always call a callback method just before starting the shutdown / kill task and another one just after the server has stopped. See at the callbacks section for more information.

4.8 ServerORB callbacks

The `ServerORB` has a callback mechanism, which simplifies the interaction between the `ServerORB` and the service.

You need to extend the class: `ServerORBEvents`. This could be the service but also any other class, which extends the `ServerORBEvents`. The subclass needs to be registered to the `ServerORB` by calling the method `setCallbacks()`.

As shown in the example above, an inner class of the service can extend the `ServerORBEvents` class.

4.9 Log

All the `ServerORB` instances will normally use a common log file. However you can redirect the log for a specific instance by passing a log group when the `ServerORB` is created. You should do this if you wish to have the `ServerORB` log messages and your service's log messages in the same file.