IP Server 2.0 Component Specification

1. Design

This component implements a server – client communication model over TCP/IP. The server is multithreaded and uses the New Java I/O API for maximum performance. The server uses only one thread to handle listening for connections, accepting and reading requests. The idea of the component is to provide TCP/IP communication services to applications without exposing them to the Java socket API or to java.nio.

The main classes of this design are IPServer and IPClient. One represents the server side and one represents the client side. The IPServer class has configuration details such as bind address, listening port and maximum number of connections. It also allows configuring who handles each type of request. The requests are handled by Handler subclasses. Finally it has methods to start and stop the server.

The IPClient class has configuration details such as the address and port of the server to connect to. It has methods for connecting, disconnecting, sending and receiving messages in a synchronous (blocking) or asynchronous (non-blocking) manner.

The Handler abstract class is a base for subclasses that handle requests. Each handler has a unique name and the requests indicate this name. The IPServer does the dispatching and calls the appropriate handler for each request. Handlers are also notified about incoming connections. A handler has a limit on the number of requests it can service simultaneously (a waiting queue).

The Connection class is a data only class that holds information about a connection.

The IPServerManager is a singleton class that can be used to manage more IP server instances. The IP server instances are identified by unique names. The usage of this manager is optional. It should only be used if needed. This manager also provides configuration file abilities. It allows setting servers with handlers up by using only a configuration file. The servers can even be started automatically.

The KeepAliveHandler class implements a Handler that responds to keep-alive message (simply sends the request message back). The server side does not need any other modifications to support keep-alive functionality. All the user has to do is add an instance of this class to the IP server.

The KeepAliveIPClient implements an IP client that sends keep-alive messages on configurable intervals. It simply intercepts the connect and disconnect calls (by overriding and proxying) and sets up using the Heartbeat component the periodic execution of a method that sends and receives keep-alive methods.

Changes in 2.0:

The major change in IP Server 2.0 is the change of message handling. New interfaces and classes are introduced in order to adopt different serialization methods easily. The serialization method can be Java's serialization framework, Serialization 1.0 component, or any customized serialization algorithm.

Since a message factory is provided, application should avoid directly create a message instance by calling the constructor. Instead, use the message factory to create the message. This can provide flexibility when new message replaces the old one without modifying the code.

The Message interface is used to send data between client and server. This can be a request or a response or something else, depending on the semantics defined by the application using it. The Message interface also contains a method which defines which MessageSerializer implementation is used to serialize/deserialize the message. An abstract Message implementation is not provided since serialization algorithm may require all super classes to implement certain interface (e.g. java.io.Serializable).

The MessageSerializer interface is used to serialize/deserialize a group of Message implementations, such as all Message implementations implementing java.io.Serializable interface. It contains two methods which serialize and deserialize the message. Implementations should be stateless.

The MessageFactory interface is a factory class managing a map of names and message types. It can create messages according to the name or a byte array by deserializing the data. It can also serialize a given message into a byte array.

The DefaultMessageFactory class implements MessageFactory interface. It provides the feature to load the name-message mapping from configuration. It also provides the methods to modify the mapping programmatically. When serializing the data, it adds class name of the MessageSerializer implementation in order to quickly identify the proper serializer when deserializing the data. It also keeps all used MessageSerializer instances in a map in order to avoid redundant instantiation via reflection.

The SimpleSerializableMessage class implements Message interface to provide a basic implementation. It contains only handler ID and request ID without any other data. This class also implements java.io.Serializable interface, thus it can be the base class of all messages implements java.io.Serializable. It uses SerializableMessageSerializer class as the serializer.

The SerializableMessageSerializer class implements MessageSerializer interface to provide the functionality of serializing any Message implementation that implements java.io.Serializable. It uses ObjectInputStream and ObjectOutputStream to fulfill the requirement.

In order to handle partially received message, the length of the serialized message should be transmitted before the message itself. When sending or receiving messages, the first four bytes denote the length of message in a Big-Indian order.

1.1 Design Patterns

The singleton pattern is used by the manager class because there is no reason to have multiple managers. It also provides an easy access point for the users.

The data value object pattern is used in the Connection. This class facilitates the data exchange inside the component.

The strategy pattern is used in the Handler abstract class, Message interface and MessageSerializer interface. Handler abstract class is used to abstract the processing of the client requests. Note that the connection notification corresponds more to a listener pattern. Message interface is used to carry various data between clients and servers. MessageSerializer interface is used to serialize/deserialize a message.

The factory pattern is used in the MessageFactory interface. It is used to create/serialize/deserialize messages with in a class.

1.2 Industry Standards

TCP/IP, sockets, Java New I/O.

1.3 Required Algorithms

1.3.1 Loading the configuration file for IPServerManager

The default configuration manager file is loaded only if the configuration manager namespace is not created when the component is instantiated for the first time. This gives the flexibility to use other configuration manager file locations or formats, essential in development of applications using many different components, with different configuration files, to avoid the nightmare of having a lot of different configuration files, spread everywhere. The configuration file loading algorithm is the following:

```
 read server names ("servers")
 for each server_name in server names
    read server address (server_name + "_address") (optional)
    read server port (server_name + "_port")
    read server max connections (server_name + "_max_connections") (0 if missing)
    read namespace used to create message factory
        (server_name + "_message_factory_namespace")
    read server backlog (server_name + "_backlog") (0 if missing)
    read server started (server_name + "_started")
    read server handler names (server_name + "_handlers")
    create IPServer instance with read port, max connections and factory namespace
    for each handler_name in handler names
       read handler class (server_name + "_" + handler_name + "_class")
       read   handler   max   requests   (server_name   +   "_"   +   handler_name   +
"_max_requests")
          (0 if missing)
       create handler using reflection (Class.forName, Class.newInstance)
       add handler to server
    add server to map
    if server started property is true, start it.
```

1.3.2 *Setting up the server for listening*

The following code can be used to set up the listening socket:

```
// create the server socket channel
ServerSocketChannel server = ServerSocketChannel.open();
// set non-blocking I/O
server.configureBlocking(false);
// set listening port
If (address == null){
   Address = InetAddress.getLocalHost().getHostName();
}
server.socket().bind(new InetSocketAddress(address, port), maxConnections);
// create the selector
Selector selector = Selector.open();
// recording server to selector (type OP_ACCEPT)
server.register(selector, SelectionKey.OP_ACCEPT);
started = true;
shouldStop = false;
new Thread(this).start();
```

1.3.3 *Accepting connections and receiving request*

The following code can be adapted for accepting connections and receiving requests and a scalable manner, using java.nio features. Note that at least the error handling from this sample code must be improved:

```
try {
    while (!shouldStop) {
            // waiting for events
            selector.select(1000);
            // get keys
            Set keys = selector.selectedKeys();
            Iterator it = keys.iterator();

            // for each keys...
            while (it.hasNext()) {
                    SelectionKey key = (SelectionKey) it.next();

                    // Remove the current key
                    it.remove();
```

```
                   // if isAcceptable = true then a client required a connection
                   if (key.isAcceptable()) {
                       if ((maxConnections > 0) && (connectionIdToChannel.size() >
maxConnections) {
                               continue;
                           }
                          // get client socket channel
                          SocketChannel client = server.accept();
                          // Non Blocking I/O
                          client.configureBlocking(false);
                          // recording to the selector (reading)
                          client.register(selector, SelectionKey.OP_READ);

                          Connection connection = new Connection(
                                     generator.getNextUUID().toString(),
                                     this, client.socket());
                          connectionIdToChannel.put(connection.getId(), client);
                          channelToConnection.put(client, connection);

                          // Notify all handlers onConnect(connection)

                           Iterator connIter = this.handlers.values().iterator();

                           while(connIter.hasNext()) {

                                ((Handler) connIter.next()).onConnect(connection);

                           }

                   // if isReadable = true

                   // then the server is ready to read
                   } else if (key.isReadable()) {

                          final SocketChannel client = (SocketChannel)
key.channel();

                          // read request message
                           // Read byte coming from the client, see 1.3.5
                           ByteBuffer buffer = readMessage(client);


                          // transfer buffer to Message obejct
                          Handler handler = getHandler(request.getHandlerId());
                          If (handler == null) {
                             Continue;
                          }
                          Connection connection = (Connection)
channelToConnection.get(client);
                          handler.handleRequest(connection, request);
                   }
           }
    }
} catch (Exception e) {
} finally {
    started = false;
    // close everything
```

```
}
```

1.3.4 *Writing messages through TCP/IP*

The following code can be used to write a message to a SocketChannel:

```
// Wraps byte array size and byte array contents into a byte buffer array
byte[] bytes = messageFactory.serializeMessage(request);
ByteBuffer lengthBuffer = (ByteBuffer)
ByteBuffer.allocate(4).putInt(bytes.length).flip();

// Wrap request bytes
ByteBuffer buffer = ByteBuffer.wrap(bytes);
this.sChannel.write(new ByteBuffer[] {lengthBuffer, buffer})
```

1.3.5 *Reading message from TCP/IP*

The following code can be used to read a message from a SocketChannel:

```
// Read byte coming from the client
ByteBuffer buffer = ByteBuffer.allocate(4);
client.read(buffer);
buffer.flip();
int size = buffer.getInt();
buffer = ByteBuffer.allocate(size);
client.read(buffer);
buffer.flip();
// ignore badly formatted requests (to add code)
Message request = messageFactory.deserializeMessage(buffer.array());
```

1.3.6 *Limiting number of simultaneously handled requests*

The Handler.handleRequest method is supposed to implement a queue mechanism for the request handling in the onRequest method. The algorithm is:

```
- enqueue (connection, request)
- if currentRequests < maxRequests then
        currentRequests++
        spawn new thread
```

The spawned threads do the following:

```
- while queue is not empty
-       dequeue (connection, request)
-       call onRequest(connection, request) (implemented by each subclass).
        currentRequests—
        connection.getIPServer().closeConnection(connection.getId());
```

This mechanism will ensure that the number of threads running onRequest simultaneously will be within the maxRequests limit.

Note that any access to the counter (get value and increment, decrement) is done inside a synchronized(this) block. Warning, get value and increment must be executed atomically.

1.3.7 *Connecting to the server*

The following code can be used in the client class to connect to the server:

```
this.sChannel = SocketChannel.open();
```

```
/**
 * Used for blocking implementation. We not use Channel's blocking, for it will block forever for the read
 * method.
 */
this.selector = Selector.open();

// configure SocketChannel with blocking style to ensure connect successful
this.sChannel.configureBlocking(true);

this.sChannel.connect(new InetSocketAddress(address, port));

// Register selector only under non-blocking mode
this.sChannel.configureBlocking(false);
this.sChannel.register(selector, SelectionKey.OP_READ);
```

1.3.8 Queuing responses

The IPClient.receiveResponse method reads from the server a response given the request id. If the response queue is not empty, a message is looked up there first. If the queue is empty or no message was found, messages are read from the server until one is found. In non-blocking mode messages are read but only as long as they are available. If a message is read but does not match the requestId, it is put in the queue for future receiveResponse calls. The algorithm is the following:

```
 Message result  = (Message) responses.get(requestId);
 if (result != null) {
        return result;
}
while (true) {
        channel.configureBlocking(blocking);

        // Read byte coming from the client
        ByteBuffer buffer = ByteBuffer.allocate(4);
        int read = channel.read(buffer);
        if (blocking & read <= 0) {
               return null;
        }
        channel.configureBlocking(false);
        channel.read(buffer);
        buffer.flip();
        int size = buffer.getInt();
        buffer = ByteBuffer.allocate(size);
        channel.read(buffer);
        buffer.flip();

        // ignore badly formatted requests (to add code)
        Message message = messageFactory.getMessage(buffer.array());
        if (message.getRequestId().equals(requestId)) {
               return message;
        }
        responses.put(message.getRequestId(), message);
}
```

1.3.9 Serializing message via MessageFacotry

```
// Get serializer class name from the message
String serializerClassName = message.getSerializerType();

// Get the serializer from the map
MessageSerializer serializer = serializerMap.get(serializerClassName);

// If the serializer is new
If(serializer == null) {
    // Create a new instance via reflection and add it in the map
    serializer = Class.forName(serializerClassName).newInstance();
    serializerMap.put(serializerClassName, serializer);
}

// Create output streams
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream output = new DataOutputStream(baos);

// Write the class name first
output.writeUTF(serializerClassName);

// Serialize message
serializer.serializeMessage(message, output);

// Close stream and return data
output.close();
return baos.toByteArray();
```

1.3.10 Deserializing message via MessageFactory

```
// Create input streams
ByteArrayInputStream bais = new ByteArrayInputStream(data);
DataInputStream input = new DataInputStream(bais);

// Read the serializer class name from the stream
String serializerClassName = input.readUTF();

// Get the serializer from the map
MessageSerializer serializer = serializerMap.get(serializerClassName);

// If the serializer is new
If(serializer == null) {
    // Create a new instance via reflection and add it in the map
    serializer = Class.forName(serializerClassName).newInstance();
    serializerMap.put(serializerClassName, serializer);
}

// Deserialize the message
Message message = serializer.deserializeMessage(input);

// Close the stream and return data
input.close();
return message;
```

1.4 Component Class Overview

IPServerManager:

The IPServerManager class is used to manage the instances of the IPServer class. This is the main usage of this class, containing named IPServer instances. Note that IP server names are case sensitive. Methods to add, remove, clear and get IP server instances are provided to fulfill this task.

Using this class is not required. The user can use IP server instances without registering them with the manager. This class is only provided as a convenient way to keep and access the IP server instances inside an application.

This class has also the role of handling instantiation and configuration of the IP server implementations using a configuration file, without writing any code. The user can either configure everything through code using the provided API or can simply write it all in a configuration file. The manager reads the configuration file and dynamically creates instances of the IP servers. It also creates the handlers for the IP server using reflection.

This class is a singleton. There is no reason in having multiple managers since multiple IP server instances can be defined.

IPServer:

The IPServer class represents the server-side of this component.

This class implements a non-blocking server that uses modern java.nio features for socket connection management instead of the classic multithreaded techniques. This allows far greater scalability because the socket accepting and request reads are done using a single thread (a non java.nio implementation will use one plus number of accepted connections threads).

Note that this does not mean threads aren't used at all. The java.nio takes care that everything that is socket related, including request reading, is done using one thread. But the actual request processing must be executed multithreaded. The number of threads per handler can be configured at the Handler level.

The server only takes care of reading requests and writing responses. The actual requests are handled by the Handler subclasses. The server dispatches the requests to the handlers according to the handler configuration.

The API exposed by this class allows server configuration (host, port, maximum number of connections), request handler configuration and methods to start and stop the server.

Connection:

The Connection class encapsulates information about a client connection on the server side. This class is only used to provide the user with convenient information about the client connection.

Message:

The Message interface represents the messages that are sent between client and server.

A message can be either a request (client to server) or response (server to client). But this only applies to a typical client server architecture where the server responds to requests. It can be the other way around in some situations or even mixed. For example, the client connects and the server sends the id yourself request.

For this reason and because requests and responses have the same structure, we have a general Message class instead of two Request and Response classes.

SimpleSerializableMessage:

It is a simple implementation of Message interface which carries no data. It implements the java.io.Serializable interface and use SerializableMessageSerializer to serialize and deserialize. It also serves as a base class of all messages using Java's serialization algorithm. An abstract Message implementation implements additional interface is not provided, since usually serialization framework requires all super classes to implement certain interface.

MessageSerializer:

The MessageSerializer interface encapsulates the algorithm of serialization and deserialization for certain classes. The serialization and deserialization algorithm in one implementation must be a pair, i.e. the data serialized by one implementation must be able to deserialized by the same instance. This class is used on both server and client side. With this interface, custom serialization can be easily adopted to existing applications.

The reason why serialization is not integrated into Message is that, one serialization algorithm may apply to a group of Message implementations. If the serialization is in the Message implementation itself, redundancy may exist.

SerializableMessageSerializer:

This is an implementation of MessageSerializer interface to provide the functionality of serializing Message instances using Java serialization algorithm. All messages serialized by this class must implement java.io.Serializable interface as required by Java serialization framework. It is also the serializer used by SimpleSerializableMessage class.

MessageFactory:

MessageFactory interface provides a single entry for common tasks which are performed on a Message instance. The tasks include creating a new Message instance, deserializing Message data and serializing a Message instance. The creation of a new Message instance is done by a given name representing the Message implementation, the request ID and the handler ID. It also provides transparent serialization and deserialization process by automatically locating proper MessageSerializer implementation.

DefaultMessageFactory:

DefaultMessageFactory is the default MessageFactory implementation. It maintains a mapping between names and Message implementations. Initially, the mapping is loaded from the configuration, either from the default namespace 'com.topcoder.processor.ipserver.message', or from a given namespace. The mapping can be programmatically modified and queried via the provided methods. When serializing the data, information identifying the MessageSerializer class is added. Such information is used to quickly identify proper MessageSerializer when deserializing.

Handler:

The handler class abstracts the server side notification and request processing API. The notifications cover successful connection and request arrival. The request arrival is not only a notification method because the implementations are expected to respond to it.

This class also defines a maximum connection number, indicating how many requests can be handled simultaneously. The meaning of this number is that at most that many threads can be executing the onRequest method at one time. If more requests arrive, then they are queued until a thread is available to handle it.

The users are expected to subclass this class and override some of or all three notification methods (onConnect, onRequest). The default implementations of these methods do nothing (except argument validation).

IPClient:

The IPClient class represents the client side of this component. This class contains method that can be used by the client side to communicate with the server.

There are configuration methods, connect, disconnect and connection query methods and methods for sending and receiving messages in a synchronous or asynchronous manner (blocking or non-blocking).

This class also implements a response queuing mechanism. Since all client-server communication can be asynchronous, multiple requests can be sent before receiving any responses. Even in synchronous mode, the client can send multiple requests to server. The server may respond to them in any order (even for the same handler, because the server calls the handler using different threads). The user has a method to get a message given the requestId. But if the first available message is not for that request, it must be skipped and another read. Skipping does not mean losing it. It is simply put in the responses queue. Note that queue is not an appropriate term since there is no order for the responses. That's because the server does not guarantee any order for the responses.

KeepAliveHandler:

Handler subclass that implements keep alive handling on the server side.

For servers that need to have such capability the user will have to add an instance of this class as handler.

This class only overrides the onRequest method. It will simply send back the request message it gets.

KeepAliveIPClient:

Subclass of the IPClient class that implements a client that keeps alive a connection by sending dummy messages at regular intervals.

The class overrides the connect and disconnect methods. The connect method calls the super connect and then starts the keep alive thread in the background (using the Heartbeat component). The disconnect method calls the super disconnect and stops the keep alive thread.

The keepAlive method (called at regular intervals by Heartbeat) simply sends a dummy Message and reads the response.

1.5 Component Exception Definitions

Exception ConfigurationException:

Exception used to signal configuration related problems.

It is used to wrap any exceptions related to the configuration data or that indicate a problem with the configuration file: configuration manager exceptions indicating bad or missing file, reflection exceptions indicating bad class names. It may also be used to indicate missing properties.

Please see the sample configuration files for details.

The message passed in the constructor should be something meaningful to help the user to find the problem. In addition to the message, the actual exception (if any) is passed to the user.

Exception ProcessingException:

The ProcessingException exception is used to wrap any implementation specific exception that may occur while servicing notifications in the handler subclasses.

For example, some handler implementation may work with files or with databases. This means I/O or SQL exceptions may occur. Such failures must be reported to the user, and that's what this exception is used for.

Note that typically exceptions will be reported to the client, meaning the exception will be wrapped in a Message subclass. This exception is only meant to be used to fatal exception that should halt the IP server. The handlers should not let exceptions out. They should catch them and log them in most cases. So typically this exception will be seldom used.

Exception MessageCreationException:

This custom exception is thrown if error occurs when creating, serialzing or deserializing messages. It serves as the base exception of all custom exceptions in the com.topcoder.processor.ipserver.message package. Usually, it wraps the original cause of error.

Typically, this exception is thrown when instantiation of a Message implementation fails. It may be caused by the constructor of Message implementation throws an exception, the Message implementation is an abstract class, proper constructor cannot be accessed, etc.

Exception UnknownMessageException:

This custom exception is thrown if the name of a message type cannot be found in the MessageFactory or a proper MessageSerializer cannot be located to deserialize a Message.

Missing proper MessageSerializer can be two cases. One case is that the MessageSerializer information exists at the beginning of the serialized data, but such class either cannot be found, or cannot be instantiated. This may happen when server-side or client-side has one missing MessageSerializer implementation. The other case is that the serialized data does not contain MessageSerializer information, which is also considered unknown message.

Exception MessageSerializationException:

This custom exception is thrown when any error occurs during serializing/deserializing process.

For example, when a Message cannot be serialized/deserialized by specific implementation of MessageSerializer due to any reason, such as I/O exception, corrupted data, etc., this exception is thrown. This exception should wrap any exceptions thrown by MessageSerializer implementations except NullPointerException and IllegalArgumentException.

Exception IOException:

This exception is used all over the component and the cause will always be socket errors. This may be thrown while connecting a socket, while setting a server socket, while disconnecting, while reading and writing data and while accepting connections.

Exception IllegalStateException:

This exception is thrown by all client and server configurations methods (modifiers), sending and receiving method, start, stop, connect and disconnect method, if the connected or started status is not valid (for example configuring a started or connected instance, starting, stopping, connecting and disconnecting twice).

Exception NullPointerException:

This exception is used in all classes for null arguments. Each method that used this exception has a javadoc tag explicitly mentioning it. In some cases one of the many arguments may be null. In those cases the javadoc clearly mentions it. See the javadocs for method level details.

Exception IllegalArgumentException:

This exception is used in all classes for invalid arguments. This is used for empty names in MessageFactory, integer argument range validation (port, connection number) and when sending a message on an invalid connection id. Note that empty strings are allowed as names in IPServerManager and as ids because it is so in IP Server 1.0.
Empty string means string of zero length or string full of whitespaces.

1.6 Thread Safety

Although not explicitly stated, it is obvious that due to the nature of this component, thread safety is a requirement, especially in the server side. Here is a discussion on how to make each class thread safe.

IPServerManager – The access to the private map must be synchronized. The lock should be made on the map itself. Note that using a synchronized map implementation is not correct. The add method consists of two Map calls for example. Even if each of them is atomic, both together aren’t. The singleton instantiation must also be synchronized.

IPServer – The access to the handler map must be synchronized. The lock should be made on the map itself, same as for IPServerManager. The access to the started attribute should be protecting using a lock on the IPServer instance. The configuration methods should run inside the lock for the started attribute. The access to connectionIdToChannel map should be synchronized in run and sendResponse. When writing the channel using sendResponse, the channel itself should be synchronized.

Handler – The subclasses should implement onConnect and onRequest in a thread safe manner. The handleRequest method is delicate as thread safety is regarded. This method implements essentially a waiting queue for the onRequest method. It ensures that only the maximum number of threads can be inside the method at one time. This waiting mechanism is implemented used wait and notify. The access to the counter (currentRequests) need to be synchronized on the instance itself. Note that the test in the while and the incrementation must be executed together, atomically.

Connection – It is not mutable.

SimpleSerializableMessage – It is not mutable.

SerializableMessageSerializer – It is stateless.

DefaultMessageFactory – The access to two mappings must be synchronized carefully. serializeMessage and getMessage(byte[]) should synchronize on serializer mapping. Other methods should synchronize on message type mapping.

IPClient – The access to the connected attribute must be synchronized on the instance. The configuration methods must be executed inside the synchronized block for the connected attribute test. The access to responses must also be synchronized. Note that checking the size and taking a message must be executed atomically.

KeepAliveHandler – Nothing special is needed. It inherits thread safety from the parent.

KeepAliveIPClient – The keepAlive method should use synchronization to avoid sending keep-alive messages before the responses are received (in case the latency is greater then the keep-alive delay). The other methods inherit thread safety from the parent.

The exceptions are immutable so they are thread safe.

2. Environment Requirements

2.1 Environment

JDK 1.4 (because java.nio is used as required)

2.2 TopCoder Software Components

- Configuration Manager 2.1.3 is used for configuration purposes. This component is used by getting its singleton instance with ConfigManager.getInstance(). Then the existsNamespace method should be used to determine whether the namespace is already loaded. If not, the add method is used to load the default configuration file. Finally getString and getStringArray return the values of the properties.

- Base Exception 1.0 is used as a base class for all exceptions. The purpose of this component is to provide a consistent way to handle the cause exception for both JDK 1.3 and JDK 1.4. Although this component uses only JDK 1.4, it is still a good idea to make all TopCoder components follow a common approach.

- Heartbeat 1.0 is used in the KeepAliveIPClient class to send keep-alive messages at regular intervals. A HeartBeatManager is created in the constructor. The connect method starts the heartbeats with heartBeatManager.add(this, delay) and the disconnect method stops the heartbeats with heartBeatManager.remove(this). The KeepAliveIPClient itself implements the HeartBeat interface. After the heartbeats are started, the HeartBeat component calls the keepAlive method at the configured intervals.

- GUID Generator 1.0 is used in the IPServer class to assign unique ids to the new incoming connection. This component has the advantage of not requiring persistent storage (such as ID Generator requires), making the component easier to use. A generator is obtained with UUIDUtility.getGenerator(UUIDType.TYPEINT32). Then using generator.getNextUUID().toString() ids are generated as needed.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- Xerces 1.4.4 is used because it is an indirect dependency of Configuration Manager 2.1.3.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.processor.ipserver

com.topcoder.processor.ipserver.keepalive

com.topcoder.processor.ipserver.message

com.topcoder.processor.ipserver.message.serializable

3.2 Configuration Parameters

Configuration values for IPServerManager

Parameter	Description	Values
servers	The list of server names. **Required**	server1 server2 server3
xxx_address	The listening IP address for one server. ‘xxx’ means one server name in ‘servers’. Default is 0.0.0.0. **Optional**	192.168.0.2
xxx_port	The listening port for one server. **Required**	8080
xxx_max_connections	Maximum concurrent connections for one server. Default is 0 (unlimited). **Optional**	500
xxx_message_factory_namespace	The namespace used to create message factory for the IP server. **Required**	com.topcoder.processor. ipserver.message
xxx_started	A flag indicating whether the server is started upon startup. Default is ‘false’. **Optional**	True/false
xxx_backlog	The number of backlog for one server. 0 means 50. Default is 0. **Optional**	100
xxx_handlers	The names of handlers for this server. **Required**	handler1 handler2 keepalive
xxx_yyy_class	The qualified class name of the handler. ‘yyy’ means the handler name in ‘xxx_handlers’. **Required**	com.topcoder.Handler1
xxx_yyy_max_requests	The maximum concurrent requests for this handler. Default is 0 (unlimited). **Optional**	30

Sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<CMConfig>
    <!--Server names, required-->
    <Property name="servers">
      <Value>server1</Value>
    </Property>

    <!--Listen IP address, optional-->
    <Property name="server1_address">
      <Value>127.0.0.1</Value>
    </Property>
    <!--Listen port, required-->
    <Property name="server1_port">
      <Value>12345</Value>
    </Property>
    <!--Maximum connections, optional-->
    <Property name="server1_max_connections">
      <Value>100</Value>
    </Property>
    <!--Namespace used to create message factory, required-->
    <Property name="server1_message_factory_namespace">
      <Value>com.topcoder.processor.ipserver.message</Value>
    </Property>
    <!--Start upon startup, optional-->
    <Property name="server1_started">
      <Value>false</Value>
    </Property>
    <!--Number of backlog, optional-->
    <Property name="server1_backlog">
      <Value>100</Value>
    </Property>
    <!--Names of handlers, required-->
    <Property name="server1_handlers">
      <Value>handler1</Value>
    </Property>

    <!--Class name of handler, required-->
    <Property name="server1_handler1_class">
      <Value>com.topcoder.processor.ipserver.keepalive.KeepAliveHandler</Value>
    </Property>
</CMConfig>
```

Configuration values for MessageFactory

Parameter	**Description**	**Values**
MessageTypes	The map between names and Message class names. **Required**	none
MessageTypes/xxx	The name and qualified class name of Message implementation. 'xxx' is the name, and the value is the qualified class name. **Required**	com.topcoder.Message1

Sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<CMConfig>
    <!-- The mapping between names and message class names, required -->
    <Property name="MessageTypes">

      <!-- Property name is the name, property value is the class
        name.There must be at least one of this property. -->
      <Property name="simple">

<Value>com.topcoder.processor.ipserver.message.serializable.SimpleSerializableM
essage</Value>
      </Property>
      <Property name="KeepAlive">

<Value>com.topcoder.processor.ipserver.message.serializable.SimpleSerializableM
essage</Value>
      </Property>

    </Property>
</CMConfig>
```

3.3 Dependencies Configuration

Follow the configuration of dependent components.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Load the configuration before using this component. Follow the demo.

4.3 Demo

The sample configuration files in 3.2 are used here.

4.3.1 Server side usage

```
Manage IP servers

// get manager instance
IPServerManager manager = IPServerManager.getInstance();
// add an IP server
manager.addIPServer("serv1", new IPServer(null, 8080, 0));
// remove an IP server
manager.removeIPServer("serv1");
// clear all IP servers
manager.clearIPServers();
// check if an IP server is registered
boolean contained = manager.containsIPServer("serv1");
// get the names of all registered IP servers
Set serverNames = manager.getIPServerNames();
// get an IP server by name
IPServer server = manager.getIPServer("serv1");

Create and configure IP servers

// create an IP server
server = new IPServer("10.0.0.255", 80, 10);
// get its address
String address = server.getAddress();
// get its port
int port = server.getPort();
// get the maximum number of connections
int maxConn = server.getMaxConnections();
// set bind address
server.setAddress(address);
// set listening port
server.setPort(port);
// set maximum number of connections
server.setMaxConnections(maxConn);

Define a custom handler

// custom handler implementation
class MyHandler extends Handler {
    MyHandler(int maxRequests) {
        super(maxRequests);
    }

    // on connect print a message
    protected void onConnect(Connection conn) {
        System.out.println(conn.getClientNameAddress()
                + " " + conn.getClientIPAddress()
                + " " + conn.getClientPort());
    }
    // on request send back a response
    protected void onRequest(Connection conn, Message request) throws IOException
{
        // process request ...
        Message response = conn.getIPServer().
                               getMessageFactory().
                               getMessage("simple",request.getHandlerId(),
                                                       request.getRequestId());
        conn.getIPServer().sendResponse(conn.getId(), response);
    }
}
```

Manage handlers

```
// add the custom handler with 10 simultaneous requests
server.addHandler("h1", new MyHandler(10));
// add a keep alive handler
server.addHandler(KeepAliveHandler.KEEP_ALIVE_ID, new KeepAliveHandler());
// remove a handler
server.removeHandler("h1");
// clear all handlers
server.clearHandlers();
// check if a handler is contained
contained = server.containsHandler("h1");
// get the ids of all handlers
Set handlerIds = server.getHandlerIds();
// get a handler by id
Handler handler = server.getHandler("h1");
```

Start and stop a server

```
// start the server
server.start();
// check the status of the server
boolean started = server.isStarted();
// stop the server
server.stop();
```

4.3.2 Client side usage

Create and configure client

```
// create a client
IPClient client = new IPClient("192.168.0.1", 139);
// create a client with keep alive capabilities that pings the
// server every 10 seconds to keep the connection alive
IPClient client2 = new KeepAliveIPClient("192.168.0.2", 445, 10000);

// get the address to connect to
address = client.getAddress();
// get the port to connect to
port = client.getPort();
// set the address to connect to
client.setAddress(address);
// set the port to connect to
client.setPort(port);
```

Connect and disconnect client

```
// connect to server
client.connect();
// check connected status
boolean connected = client.isConnected();
// disconnect from server
client.disconnect();
```

Implement a custom message

```
// custom message implementation
```

```
class MyMessage extends SimpleSerializableMessage {
    private String data = null;

    MyMessage(String handlerId, String requestId) {
        super(handlerId, requestId);
    }

    synchronized void setData(String data) {
        this.data = data;
    }

    public synchronized String getData() {
        return data;
    }
}
```

Send requests and receive responses

```
// send three requests with different request ids
MyMessage message;
message = (MyMessage) messageFactory.getMessage("mymessage", "myhandler",
               "req1112");
message.setData("some_data");
client.sendRequest(message);

message = (MyMessage) messageFactory.getMessage("mymessage", "myhandler",
              "req1115");
message.setData("some_data2");
client.sendRequest(message));

message = (MyMessage) messageFactory.getMessage("mymessage", "myhandler2",
              "req1145");
message.setData("some_data3");
client.sendRequest(message);

// get the response for request req1115 (blocks until it arrives)
Message response1 = client.receiveResponse("req1115", true);
//get the response for request reqxxxx (returns null if not immediately available)
Message response2 = client.receiveResponse("reqxxxx", false);
//get any response (blocks until one arrives)
Message response3 = client.receiveResponse(true);
//get any reponse (returns null if not immediately available)
Message response4 = client.receiveResponse(false);
```

Multiple clients, Each IPClient will send a message to the same handler

```
IPClient[] clients = new IPClient[HANDLER_NUM];

for (int i = 0; i < clients.length; ++i) {
    clients[i] = new IPClient(helper.getAddress(), helper.getPort());
    clients[i].connect();

    // send a request to server
    Message src = messageFactory.getMessage("simple", "handler0", "request" + i);
    clients[i].sendRequest(src);
    assertNotNull("Fails to recieve response.", clients[i].receiveResponse(true));
    clients[i].disconnect();
}
```

message to different handlers from a single client

```
IPClient client = new IPClient(helper.getAddress(), helper.getPort());
client.connect();

Message[] msgs = new Message[HANDLER_NUM];

for (int i = 0; i < msgs.length; ++i) {
    msgs[i] = messageFactory.getMessage ("simple", "handler" + i, "request" + i);
    client.sendRequest(msgs[i]);
}

// retrieve all the messages (from the last one to the first one).
for (int i = 0; i < msgs.length; ++i) {
    int id = msgs.length - 1 - i;
    Message ret = client.receiveResponse("request" + id, true);
    assertEquals("Fails to receive response", msgs[id].getRequestId(), ret.getRequestId());
}

client.disconnect();
```

4.3.3 Managing MessageFactory

```
// Create a MessageFactory instance
MessageFactory messageFactory =
       new DefaultMessageFactory(MESSAGE_FACTORY_NAMESPACE);
// get message
Message message = messageFactory.getMessage("simple", "handler id", "request id");
assertEquals("handler id", message.getHandlerId());
assertEquals("request id", message.getRequestId());
assertEquals("com.topcoder.processor.ipserver.message.serializable.SerializableMessageSerializer",
        message.getSerializerType());

// serialize message
byte[] data = messageFactory.serializeMessage(message);
// deserialize message
Message msg = messageFactory.deserializeMessage(data);
// msg and message are same
assertEquals(msg.getHandlerId(), message.getHandlerId());
assertEquals(msg.getRequestId(), message.getRequestId());
assertEquals(msg.getSerializerType(), message.getSerializerType());

// add message type
((DefaultMessageFactory) messageFactory).add("new type", CustomMessage.class);

// remove message type
((DefaultMessageFactory) messageFactory).remove("new type");

// get message type
Class cls = ((DefaultMessageFactory) messageFactory).get("simple");

// test if there's a certain message type
boolean res = ((DefaultMessageFactory) messageFactory).contains("new type");

// get the mapping of message type names and message types
Map all = ((DefaultMessageFactory) messageFactory).getMessageTypes();
```

```
// clear all message types
 ((DefaultMessageFactory) messageFactory).clear();
```

5. Future Enhancements

Connection tunneling is very useful when the network access is restricted by firewall or other means. Common tunneling techniques include proxy (HTTP or Socks proxy), Virtual Private Network, and customized wrapping protocols. To implement this, IPClient may use strategy pattern.

Another possible enhancement is to make the Message an independent component, and add more serialization algorithms, such as XML serialization. By this, SOAP client/server can be easily implemented.