



Terminal Connection Pooling 3270 Pathway Software

Technical Whitepaper
March 2010

Copyright © 2010 Clarity Solutions, Inc.
All rights reserved.

9930 Derby Lane, Suite 202 • Westchester, IL 60154
Phone 1-888-2-REHOST (or 1-630-981-6100)

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Clarity Solutions, Inc. and its licensors, if any.

Clarity and UniKix are trademarks, or registered trademarks, of Clarity Solutions, Inc. in the United States and other countries.

IBM and CICS are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. All other marks are the property of their respective owners.

Table of Contents

Chapter 1	Introduction.....	4
	The Problem	4
	The Solution	4
Chapter 2	3270 Connection Pooling.....	6
	Customer Information Example	6
	Basic Implementation.....	6
	Simple pooling usage	7
	Dealing with problems.....	8
Chapter 3	TerminalPool.....	10
	Commons-pool 101	10
	Implementing TerminalPool.....	11
	CustomerInformationTerminalFactory.....	11
	TerminalPoolImpl.....	12
	Pools that require authentication.....	14
Chapter 4	Conclusion	15
	Additional Information.....	15

Chapter 1

Introduction

With ever-increasing numbers of new applications being developed and deployed, along with the continued expansion of the Internet, there is a growing need for Web-based applications to be able to interact with existing mainframe applications and data. Mainframe rehosting and migration solutions from Clarity Solutions, Inc. (Clarity) provide a variety of ways to modernize legacy environments without lengthy rewriting or major enterprise change.

With Clarity™ 3270 Pathway software, new Java technology-based applications can be rapidly developed which interact with existing UniKix™ Transaction Processing (TPE) software and/or mainframe IBM® CICS® applications. Clarity 3270 Pathway software leverages the universal 3270 interface already present in the majority of mainframe transaction processing environments to provide access to almost any back-end mainframe application. Using Clarity 3270 Pathway software, developers can connect to a 3270 end system, run an application, and extract data from the end system. Data can then be used in an application written in the Java programming language.

On the client side, new integration applications can be created which access multiple back-end applications on multiple hosts concurrently and present the results in a single user interface. Traditional 3270 green screens can be replaced with easy-to-use GUI interfaces to improve end user experiences and productivity. Clarity 3270 Pathway software also enables Java Integrated Development Environments (IDEs) and Java technology-based application servers to create new user interfaces for legacy applications.

The Problem

An integration application gains access to 3270 information by connecting to one or more backend systems. It navigates through a variety of 3270 screens, inputting and extracting data where necessary, and finally disconnects from the system when it is done. While Clarity 3270 Pathway technology provides a simple and intuitive API for performing that navigation and data manipulation, using 3270 terminals in this raw form does, however, ignore the practical issues involved in the production of a commercial strength software solution.

The special issues involved in commercial strength 3270 access include:

- *Dealing with a large number of concurrent connection attempts to the 3270 system*
This leads to a large number of concurrent connections, with which the 3270 system may be unable to cope with effectively.
- *Enabling functionality without a high performance penalty*
The performance cost of establishing the connection and logging on to the system can be very high in comparison with the rest of the application.

The Solution

Internet applications which interact with databases and other resources typically use a connection pooling mechanism to avoid the potential issues previously highlighted. In most cases, connection

pooling is done transparently for the application by its container. Unfortunately, in the case of 3270 data, there is no automatic pooling available.

The remainder of this document examines 3270 terminal connection pooling by using a typical example and working through the ideas and techniques for creating an effective connection pooling infrastructure to use in conjunction with Clarity 3270 Pathway software environments.

Chapter 2

3270 Connection Pooling

There are typically two types of 3270 interaction required by a 3270 integration application. These are most notably:

1. Read-only work that does not require any specific user authentication
2. Update work that requires specific user authentication

These two types of interactions require slightly different treatment from a pooling perspective, but the same techniques apply to both. We will first examine the read-only case as this provides the best optimization, and then move on to the update case.

Customer Information Example

You have an application which needs to obtain data stored in a 3270 system and present that data to the user via a web page.

Basic Implementation

A Java application was created to obtain information from data stored in a 3270 based system. An example of this code can be seen below:

```
public CustomerInfo getCustomerInfo(String accountNumber) {
    final CustomerInfo ci = new CustomerInfo();
    // Create and connect the terminal
    final Terminal terminal = new Terminal();
    terminal.setTN3270Host(host);
    terminal.setTN3270Port(port);
    terminal.connect();
    terminal.waitForConnected();
    // Log In
    terminal.pressEnter(); // On the welcome screen
    terminal.typeString("steve"); // UserName
    terminal.typeString("stevepw"); // Password
    terminal.pressEnter();
    terminal.waitForKeyboardUnlocked();
    // Go to the customer query page
    terminal.pressPF(2);
    terminal.waitForKeyboardUnlocked();
    // Issue the query
    terminal.typeString(accountNumber);
    terminal.pressEnter();
    terminal.waitForKeyboardUnlocked();
    // Extract the data
    final int surnameOffset = terminal.offsetFromRowColumn(5, 2);
    customerInfo.setSurname(terminal.getReadableString(surnameOffset, 20).trim());
    ...
    // Navigate back out of the application
    terminal.pressPF(3);
    terminal.waitForKeyboardUnlocked();
    terminal.pressPF(3);
    terminal.waitForKeyboardUnlocked();
}
```

```

    // Disconnect
    terminal.disconnect();
    terminal.waitForDisconnected();

    return ci;
}

```

The above process is quite complex and contains a large amount of connection and navigation code. This would not be necessary if the terminal was already connected to the customer query page. In order to better see what is going on (without the detail of the precise navigations) the above code can be re-written into a set of calls to helper methods responsible for their own small parts of the process:

```

public CustomerInfo getCustomerInfo(String accountNumber) {

    final Terminal terminal = createTerminalThatIsAtTheCustomerQueryScreen();

    // Issue the query
    terminal.typeString(accountNumber);
    terminal.pressEnter();
    terminal.waitForKeyboardUnlocked();
    // Extract the data

    final CustomerInfo ci = new CustomerInfo();
    final int surnameOffset = terminal.offsetFromRowColumn(5, 2);
    customerInfo.setSurname(terminal.getReadableString(surnameOffset, 20).trim());
    ...
    // Navigate back to the customer query page
    terminal.pressPF(3);
    terminal.waitForKeyboardUnlocked();

    disconnect(terminal);

    return ci;
}

```

Now that the processing is clearer, we can begin to see how connection pooling might help in this situation. The core of the `getCustomerInfo()` method can operate on any `Terminal` that is positioned at the correct place within the application.

Simple pooling usage

Suppose that we have a pool of these `Terminal`s that are ready for work and already positioned as required within the 3270 application. Our `getCustomerInfo()` method could then be written to use this pool as follows:

```

public CustomerInfo getCustomerInfo(String accountNumber) {

    final Terminal terminal = _ciTerminalPool.borrowTerminal();

    // Issue the query
    terminal.typeString(accountNumber);
    terminal.pressEnter();
}

```

```

terminal.waitForKeyboardUnlocked();
// Extract the data
final CustomerInfo ci = new CustomerInfo();
final int surnameOffset = terminal.offsetFromRowColumn(5, 2);
customerInfo.setSurname(terminal.getReadableString(surnameOffset, 20).trim());
...
// Navigate back to the customer query page
terminal.pressPF(3);
terminal.waitForKeyboardUnlocked();

_ciTerminalPool.returnTerminal(terminal);

return ci;
}

```

We have now made use of our connection pool and our code is immediately much more efficient. It is important to note that the active parts of the `getCustomerInfo()` method do not have to worry about navigating to the correct initial 3270 screen. It must, however, worry about navigating back once the real work has been done. This is important because once the `Terminal` is returned to the pool, it can then be re-used by a subsequent operation which will expect the `Terminal` to be positioned at the correct 3270 screen.

Dealing with problems

The latest implementation of `getCustomerInfo()` also contains some serious flaws that require examination:

- *What happens if something goes wrong during terminal processing (i.e. the connection to the 3270 system terminates)?*
In this case, the `Terminal` is never returned to the pool. This situation is absolutely forbidden for pools in general. Pooled items (`Terminals` in this case) must be returned to the pool after use.
- *What happens if the 3270 system is not available at all?*
There are several mechanisms for dealing with this. First, the pool could block until a connection is ready to be re-established. This would not produce a good application since this blocking would delay the response to the initiator; possibly for a very long time. The alternative (and better) solution is to throw an exception. It is then left to the high level code to cope with this sort of problem and provide appropriate diagnostics and information to the user.

These problems are fixed by simple changes to our method as follows:

```

public CustomerInfo getCustomerInfo(String accountNumber) throws
ConnectionFailureException {

    final Terminal terminal = _ciTerminalPool.borrowTerminal();

    try {
        // Issue the query
        terminal.typeString(accountNumber);
    }
}

```

```
        terminal.pressEnter();
        terminal.waitForKeyboardUnlocked();
        // Extract the data
        final CustomerInfo ci = new CustomerInfo();
        final int surnameOffset = terminal.offsetFromRowColumn(5, 2);
        customerInfo.setSurname(terminal.getReadableString(surnameOffset,
20).trim());
        ...
        // Navigate back to the customer query page
        terminal.pressPF(3);
        terminal.waitForKeyboardUnlocked();
    } finally {
        // Ensure the terminal is ALWAYS returned to the pool
        _ciTerminalPool.returnTerminal(terminal);
    }

    return ci;
}
```

Since we now have an idea of what the correct usage of a connection pool should be, we can now look at what it takes to actually implement one.

Chapter 3

TerminalPool

Now that we have seen how a pool of `Terminals` might be used, it is time to define in more detail what such a pool is and how it works. We define a `TerminalPool` interface as one which hides all of the tricky implementation details and provides the required functionality outlined in the previous chapter. We will then look at implementations of this interface.

The `TerminalPool` solution makes use of the excellent generic pooling libraries that are part of the Jakarta Commons Project¹. The `commons-pool` library provides a comprehensive suite of facilities, of which only a few are utilized in this solution².

From the perspective of the user of the pool, the interface which it provides is very simple. Below is a definition of a pool that will be satisfactory for most real-work examples. It takes its terminology from the `commons-pool` library, but adds in specifics related to Clarity 3270 Pathway software `Terminals`.

```
public interface TerminalPool {
    Terminal borrowTerminal() throws ConnectionFailedException;
    void returnTerminal(Terminal terminal);
    void close();
}
```

The methods defined above are basically those used in the previous definition of the `getCustomerInfo()` method.

- The `borrowTerminal()` method gets a terminal from the pool.
- The `returnTerminal()` method is there to return a terminal to the pool.
- Finally, the `close()` method provides the ability to destroy the pool. This is typically used when an application terminates. For example, in a servlet environment, this would be called when an application context is destroyed. This is not used by the previous calling structure, but it is required to allow the clean integration in an application server environment.

Commons-pool 101

The `commons-pool` library essentially works with two concepts. There is the `ObjectPool` that is responsible for the pooling semantics such as when to create objects, what to do when no objects exist, etc. Then there is the `PooledObjectFactory` interface that is responsible for the precise implementation of object creation, destruction, and verification.

1 Jakarta Commons Pool is available from <http://jakarta.apache.org/commons/pool/>

2 An exploration and investigation of the `commons-pool` facilities which can enable and customize pools to precise requirements is highly recommended.

In order to create a pooling solution using `commons-pool` it is first necessary to create and configure a pool. It is then necessary to implement a factory for dealing with the details of object creation, etc.

Implementing TerminalPool

Now that we have defined the interface we want and how to use it, it is necessary to actually implement a `TerminalPool`. `TerminalPool` implementations are specific to the needs of the actual application. However, their general structure and methodology apply universally. The implementation here uses `commons-pool` as a delegate to provide the required semantics.

The key implementation class is the `GenericObjectPool`. This is the most versatile of all of the `commons-pool` implementations and provides our required level of flexibility. The `GenericObjectPool` has a large number of configurable attributes that affect the way pooling works, however this document does not detail the attributes and their meanings³. The `GenericObjectPool` operates in conjunction with a `PoolableObjectFactory`.

CustomerInformationTerminalFactory

It is this `PoolableObjectFactory` interface that is key to producing effective `Terminal` factories. Each specific implementation must perform the right operations to work together effectively. Writing a `PoolableObjectFactory`, it is necessary to implement methods that perform the operations:

- `makeObject()` In our case this is the creation of a `Terminal`, and the navigation of that `Terminal` to the base screen of the application. The base screen of the application is the one to which the `Terminal` must return when any set of operations are performed.
- `destroyObject()` In the case of 3270 access, this is the clean (or otherwise) termination of the `Terminal` connection. In the simplest (and most usual) case, all that is necessary is the disconnection of the `Terminal`.
- `validateObject()` In our case, this method needs to validate that the `Terminal` is still connected and that it is displaying the correct 'base' screen.

All of the above methods have been put together in the listing below of a `Terminal` factory specific to this application:

```
public class CustomerInfoTerminalFactory implements PoolableObjectFactory {
    private final String _host;
    private final int _port;

    public CustomerInfoTerminalFactory(String host, int port) {
        _host = host;
        _port = port;
    }

    /** {@inheritDoc} */
    public final Object makeObject() throws Exception {
        final Terminal terminal = new Terminal();
        terminal.setTN3270Host( _host );
    }
}
```

3 Refer to the `commons-pool` documentation for specific details

```

terminal.setTN3270Port( _port );
terminal.setNetworkInactivityTimeout( 0 ); // Never timeout
terminal.connect();

try {
    terminal.waitForConnected();
} catch (IllegalStateException e) {
    throw new
ConnectionFailedException(terminal.getLastDisconnectionString(), e);
}

// Note that this code is a copy of the code from the origin
getCustomerInfo method
terminal.waitForKeyboardUnlocked();
terminal.pressEnter(); // On the welcome screen
terminal.typeString("steve"); // UserName
terminal.typeString("stevepw"); // Password
terminal.pressEnter();
terminal.waitForKeyboardUnlocked();
// Go to the customer query page (base screen)
terminal.pressPF(2);
terminal.waitForKeyboardUnlocked();

return terminal;
}

/** {@inheritDoc} */
public final void destroyObject(Object obj) throws Exception {
    final Terminal terminal = (Terminal)obj;

    terminal.disconnect();
    terminal.waitForDisconnected();
}

/** {@inheritDoc} */
public final boolean validateObject(Object obj) {
    // Ensure the terminal is in a good state to perform its operations.
    final Terminal terminal = (Terminal)obj;

    if (!terminal.isConnected()) {
        return false;
    }

    // Ensure that the terminal is displaying the base screen.
    final String screenIdentifier = terminal.getReadableString(1, 70).trim();
    return screenIdentifier.equals("CUSTQUER");
}
}

```

TerminalPoolImpl

This is a complete implementation of the `TerminalFactory` for the customer query application. All that is now required is the implementation of a `TerminalPool`. Since all of the building blocks have now been identified, all is required is to glue the `GenericObjectPool` into the `TerminalPool` interface. The following code sample demonstrates this:

```

public class TerminalPoolImpl implements TerminalPool {

    private final ObjectPool _pool;

```

```

/**
 * Creates a new instance of TerminalPoolImpl.
 */
public TerminalPoolImpl(PoolableObjectFactory factory, int size) {
    final GenericObjectPool pool = new GenericObjectPool( factory );
    pool.setTestOnBorrow(true);
    pool.setTestOnReturn(true);
    pool.setMaxActive(size);
    pool.setWhenExhaustedAction(GenericObjectPool.WHEN_EXHAUSTED_BLOCK);

    // Pre-fill the pool with terminals. This improves initial
    // response times, at the cost of startup overhead.
    try {
        PoolUtils.prefill(pool, size);
    } catch (ConnectionFailedException e) {
        // Don't worry too much, this just means that the system is
        // unavailable at the moment.
    } catch (Exception e) {
        // Something else happened that is fatal to us so do something
        // appropriate
        throw new Error(e);
    }

    _pool = pool;
}

public Terminal borrowTerminal() throws ConnectionFailedException {
    try {
        return (Terminal)_pool.borrowObject();
    } catch (ConnectionFailedException e) {
        throw e;
    } catch (Exception e) {
        e.printStackTrace();
        throw new Error( e );
    }
}

public void returnTerminal(Terminal terminal) {
    try {
        _pool.returnObject(terminal);
    } catch (Exception e) {
        e.printStackTrace();
        throw new Error( e );
    }
}

public void close() {
    try {
        _pool.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

It should be noted in the previous example that a few configurations were made to the `GenericObjectPool`. These configurations perform the following roles:

- Ensure that all objects borrowed and returned are in a fit state. These settings mean that the `GenericObjectPool` calls the `validateObject()` method on the factory whenever a `Terminal` is borrowed or returned. This ensures that the user of the `TerminalPool` can be sure that what they get back is a `Terminal` in precisely the desired state.
- Limit the size of the pool to just the number specified. This ensures that the application does not consume all of the 3270 host resources by having huge numbers of connections active.
- Ensure that if all of the objects are currently in use, then the caller will block until one becomes free. This is important for our implementation to allow seamless scalability of the application.

Pools that require authentication

The previous discussions provide great efficiency improvements for situations where there are no user-specific authentication requirements. An authentication is done under the auspices of a single user and can therefore be shared by all accesses. In cases of data update, it is typically necessary for the 3270 application to execute whilst logged in as a particular user. Since there are liable to be many such users, it is not possible to pool `Terminals` on a per-user basis. Pooling therefore has to be done at a higher level.

This can easily be achieved by designating the 'base' screen of the application as the login page. This does mean that the cost of the connection process is eliminated from the transaction overhead, but does not lead to the huge savings provided by the read-only case. Unfortunately, it is the nature of the application that prevents such optimizations.

Chapter 4

Conclusion

In a real-world application, the required workload is such that multiple pools of terminals are required. At the simplest level, one pool is required for read-only work and one for work requiring authentication. In a more complex example there might be many such pools. The configuration of individual pools will differ for these applications in order to provide an adequate response profile for each part of the application suite.

The environment of the application governs how pools are accessed. In a stand-alone application, static or instance variables typically suffice for the storage and accessing of pools. In a Web application, pools can be placed into the Servlet or application context. In an EJB environment, the pools may be placed into JNDI.

In all of the above situations, the actual pool usage is identical; it is just the method for obtaining the pool that changes.

The pooling techniques shown here in conjunction with the `commons-pool` APIs allow effective terminal pooling to be used by any Java application. The performance benefits are great and well worth the effort to implement.

Additional Information

The following table identifies other sources of information related to this document.

Web Site URL	Description
www.clerity.com	Clerity's home page
www.clerity.com/products/pathway/	3270 Pathway Overview
www.clerity.com/resources/datasheets.php	Solution datasheets and whitepapers
jakarta.apache.org/commons/pool/	Jakarta Commons Pool