Profiling and Testing with Test and Performance Tools Platform (TPTP)

eclipsecon™ 2009

Speakers

Eugene Chan
IBM Canada
ewchan@ca.ibm.com

Jonathan West
IBM Canada
jgwest@ca.ibm.com

Agenda

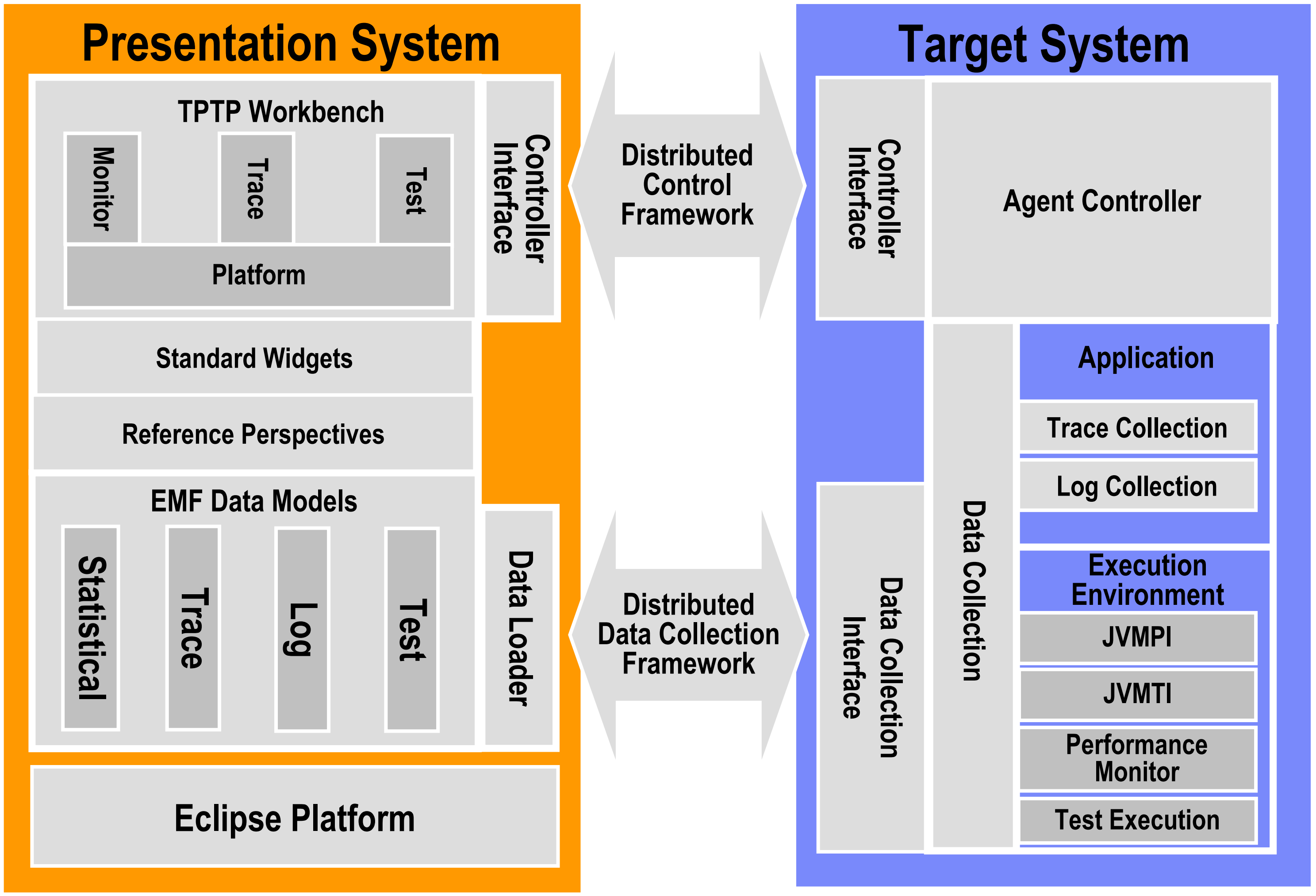
- Overview
- Architecture
- Agent Controller
- TPTP Profiling
- TPTP Testing
- Q & A

- ➢Overview
- ➢Architecture
- ➢Agent Controller
- ➢TPTP Profiling
- ➢TPTP Testing
- ➢Q & A

Overview

- 2002: Eclipse tools subproject (Hyades)
- 2004: Eclipse top-level Eclipse project (Test and Performance Tools Platform)
- Open-source platform providing an extensible framework for Automated Software Quality (ASQ) tools including reference implementations for testing, tracing, and monitoring software systems.
- Goals:
  - Platform of choice for test, performance, and monitoring tools
  - Exemplary tooling
  - Enable value-added third-party tooling through extensibility and high-quality APIs
- Composed of four sub-projects:
  - Platform : common infrastructure, models, UI frameworks.
  - Test : testing and extensible tools for specific testing environments
  - Trace and Profiling : data collection for Java and distributed applications
  - Monitoring : UI to introspect and interact with manageable resources instrumented for different types of managed resources

Architecture

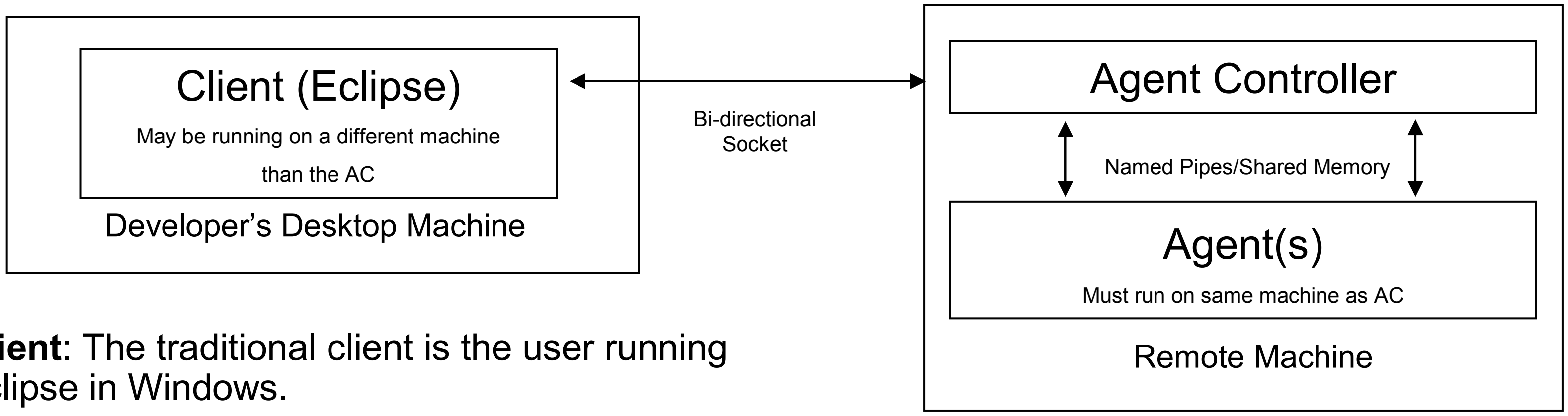


- Overview
- Architecture
- Agent Controller
- TPTP Profiling
- TPTP Testing
- Q & A

Remote Applications, Remote Agents

- Remote profiling requires remote control, and remote communication.
- When profiling an application on a remote machine, or running tests, there must be some mechanism by which remote processes are launched, and a mechanism by which data is sent back to the workbench from those remote processes.
- The TPTP solution to this problem is an **agent controller**.
- An agent controller is a separate process that sits on a remote machine, and listens for commands from the workbench. When required, it will launch agents that provide additional functionality, such as profiling or testing.

Agent Controller – Remote Scenario



Client (Eclipse)
May be running on a different machine than the AC
Developer's Desktop Machine
Bi-directional Socket
Agent Controller
Named Pipes/Shared Memory
Agent(s)
Must run on same machine as AC
Remote Machine

- **Client**: The traditional client is the user running Eclipse in Windows.
- **Agent**: A separate process that provides data to the workbench (through the agent controller). In TPTP, this is either a profiling agent, or a test agent.
- **Agent Controller**: The agent controller serves as the communication link between the client and agent.
- The general theme is: The client asks for data, or tells the Agent to start collecting data. The agent then returns the data back to the Client (through the agent controller).

- **One important rule**: *Agents will ALWAYS run on the same machine as the agent controller*. Agents can only connect to an agent controller that is running on the same machine.
- **A second important rule**: Only one agent controller may be running on a machine at a time. Additional ACs will fail to start if an existing AC is running.

Agent Controller (Standalone vs. Integrated)

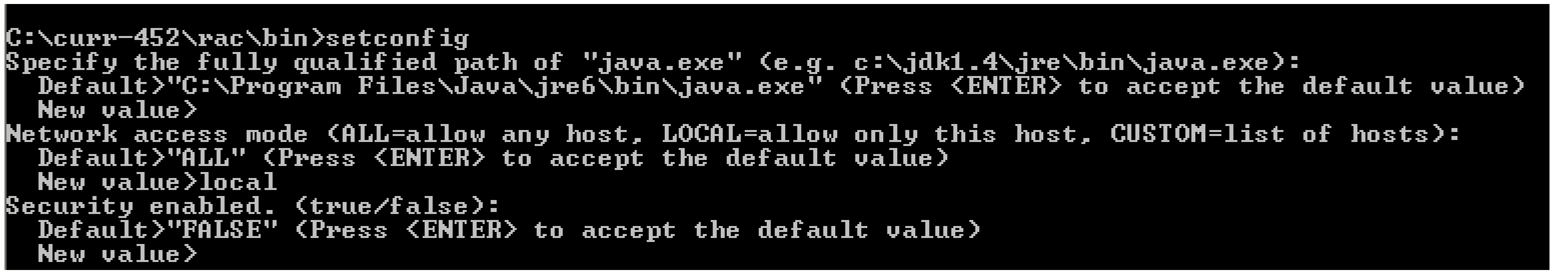
- The agent controller (AC) comes in two differing packages:
- **Integrated Agent Controller (IAC)**: When using TPTP functionality on your local machine -- and from inside Eclipse -- you do not need to download a separate package. The TPTP workbench code is packaged with an integrated agent controller (IAC) that is a pre-bundled Eclipse plug-in which will launch automatically whenever TPTP functionality is used.
- **Standalone Agent Controller**: When using TPTP functionality on a remote machine (for instance, profiling a remote web app), it is necessary to download the agent controller as a separate package **standalone** component from the TPTP workbench components. It is required that it be configured and run before remote profiling begins.
- The standalone agent controller is required for remote profiling.

Downloading the Agent Controller

- The agent controller is available for download from
  - http://www.eclipse.org/tptp/home/downloads/
- The agent controller is available for the following platforms:
  - Windows IA32
  - Windows x86-64 (EM64T)
  - Windows on Itanium
  - Linux IA32
  - Linux x86-64 (EM64T)
  - Linux on Itanium
- Presently not available for
  - Mac OSX (Bug 68111 – Help Wanted! ☺)

Installing the Agent Controller on a Remote Machine

Once Downloaded, unzip and run SetConfig:
• SetConfig.sh on Linux / SetConfig.bat on Windows (under the /bin directory of the package)
• Requires path to a JVM (will read from the path to select a default)
• Host access to the AC (ALL – Everyone, LOCAL – Local machine only, CUSTOM – Custom list of hosts).



```
C:\curr-452\rac\bin>setconfig
Specify the fully qualified path of "java.exe" (e.g. c:\jdk1.4\jre\bin\java.exe):
  Default>"C:\Program Files\Java\jre6\bin\java.exe" (Press <ENTER> to accept the default value)
  New value>
Network access mode (ALL=allow any host, LOCAL=allow only this host, CUSTOM=list of hosts):
  Default>"ALL" (Press <ENTER> to accept the default value)
  New value>local
Security enabled. (true/false):
  Default>"FALSE" (Press <ENTER> to accept the default value)
  New value>
```

- **Running from the command line**:
  - Run *ACServer* on Windows
  - Run *ACStart.sh* on Linux

- **Running as a Windows service**:
  - Run the <install-dir>\bin\manageservice.exe application to create the Windows service. The syntax is:
    - manageservice add "<service_name>" "<install-dir>“
    - For example, manageservice add "Agent Controller" "C:\tptpAC"

- Overview
- Architecture
- Agent Controller
- TPTP Profiling
- TPTP Testing
- Q & A

How Does Profiling Work

- **Profiling** is the use of tools to perform performance analysis, usually by extracting information as the target application is being executed.
- Profiling agents interface with the *Java Virtual Machine (JVM)* through a predefined interface (JVMTI for Java 1.6+/1.5, and JVMPI for 1.4.x/1.5.x).
- Using both JVM event listeners and bytecode instrumentation, the TPTP JVMTI Profiling Agent monitor Java application execution and return data related to method execution (**execution profiling**), object allocations (**heap profiling**) or thread analysis (**thread profiling**).
- **Note:** This method of profiling will necessarily slow down the execution of the program, as every single method invocation (execution profiling), or object allocation (heap profiling), will require data to be communicated back to the workbench.
- However, with an appropriate set of filters that specifically target areas of interest in your application, this slowdown can be significantly mitigated.

Environment Variables

- Before starting the application server, you'll need to set a number of environment variables.
- **Windows Environment Variables:**
  - Set TPTP_AC_HOME=*(Path to your agent controller)*
  - set JAVA_PROFILER_HOME=%TPTP_AC_HOME%\plugins\org.eclipse.tptp.javaprofiler
  - Set PATH=%JAVA_PROFILER_HOME%;%PATH%;%TPTP_AC_HOME%\bin
  - Set PATH=%PATH%;%JAVA_HOME%\bin
- **Linux Environment Variables**
  - export TPTP_AC_HOME=*(Path to your agent controller)*
  - export JAVA_PROFILER_HOME=$TPTP_AC_HOME/plugins/org.eclipse.tptp.javaprofiler
  - export PATH=$JAVA_PROFILER_HOME:$TPTP_AC_HOME/bin:$PATH:
  - export LD_LIBRARY_PATH=$JAVA_PROFILER_HOME:$TPTP_AC_HOME/lib:$LD_LIBRARY_PATH
- The easiest way of doing this is to place these environment variables inside Tomcat's startup.bat/startup.sh.

Environment Variables -- Continued

- In addition to the inclusion of these environment variables, you'll also need to set JVM options to allow the TPTP JVMTI profiling agent to interface with JVM. Include the following lines in the startup.bat / startup.sh.
- **Windows**:
  - set JAVA_OPTS=-agentlib:JPIBootLoader=JPIAgent:server=enabled;<Profile-Option>
- **Linux**:
  - export JAVA_OPTS=-agentlib:JPIBootLoader=JPIAgent:server=enabled;<Profile-Option>
- Profiling Options:
  - You'll need to select a profiling option, and place that in the <ProfileOption> field above. The following profiling option are available, based on what you wish to profile:
  - **CGProf**: This profiling option is used for identifying performance bottlenecks, by breaking down execution time at the per-method level.
  - **HeapProf**: This option allows you to identify the contents of the heap by tracking object allocation and deallocation throughout the lifetime of the program.
  - **ThreadProf**: This profiling option allows you to trace thread usage throughout the lifetime of the program.
- Example:
  - set JAVA_OPTS=-agentlib:JPIBootLoader=JPIAgent:server=enabled;HeapProf (Heap profiling on Windows)
  - OR, export JAVA_OPTS=-agentlib:JPIBootLoader=JPIAgent:server=enabled;CGProf (Execution time profiling, on Linux)

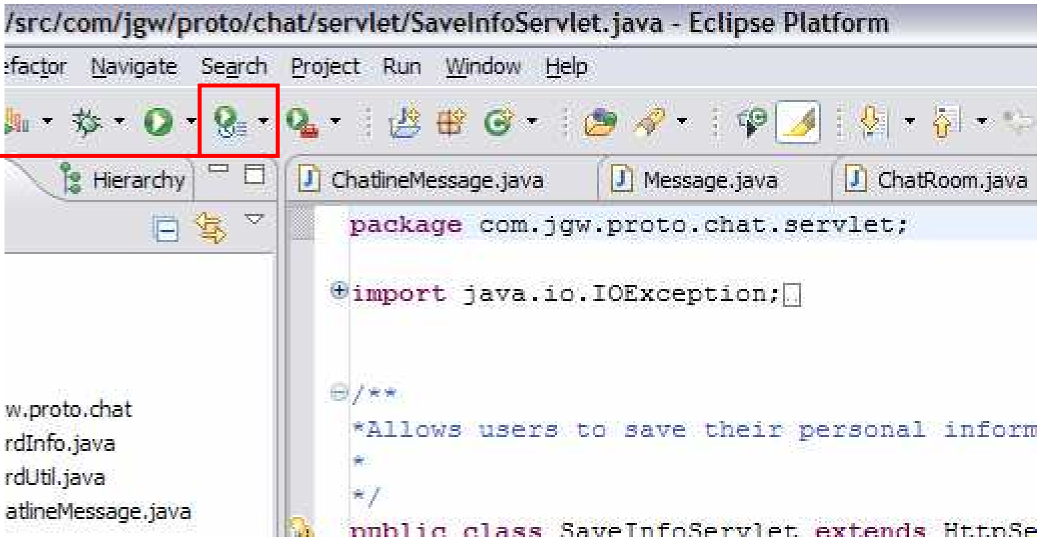
Starting up the Application Server

- Once the appropriate environment variables are in place, merely running startup.bat / startup.sh will allow you to begin profiling.
- As soon as the application server has started, you will be able to connect to it from inside Eclipse, and begin profiling your application.

Attaching to your Application Server

- Once you have installed the appropriate TPTP plugins into your workbench, you can attach to the application server for profiling using either the following icon.

- You can also access the profiling dialog through Run -> Profile Configurations.


/src/com/jgw/proto/chat/servlet/SaveInfoServlet.java - Eclipse Platform
efactor Navigate Search Project Run Window Help
Hierarchy
ChatlineMessage.java
Message.java
ChatRoom.java
w.proto.chat
rdInfo.java
rdUtil.java
atlineMessage.java

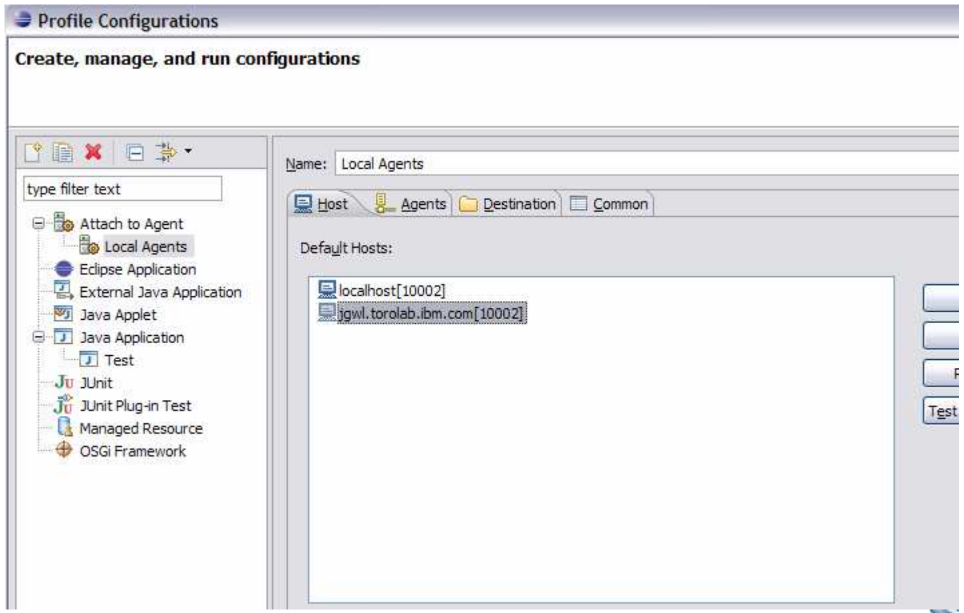
```
package com.jgw.proto.chat.servlet;

import java.io.IOException;

/**
 *Allows users to save their personal inform
 *
 */
public class SaveInfoServlet extends HttpSe
```

Profiling Perspective – Attach to Agent

- Once inside the Profile Configurations Dialog create an ‘Attach to Agent’ entry. Then, in the Hosts tab, add the remote hostname to the host list. This is how one specifies the host to connect.

- Next, switch to the agents tab. You should see an agent corresponding to the profiling option you specified in the Tomcat environment va

- Execution Time Analysis corresponds to the CGProf profiling option.



Profile Configurations

Create, manage, and run configurations

type filter text

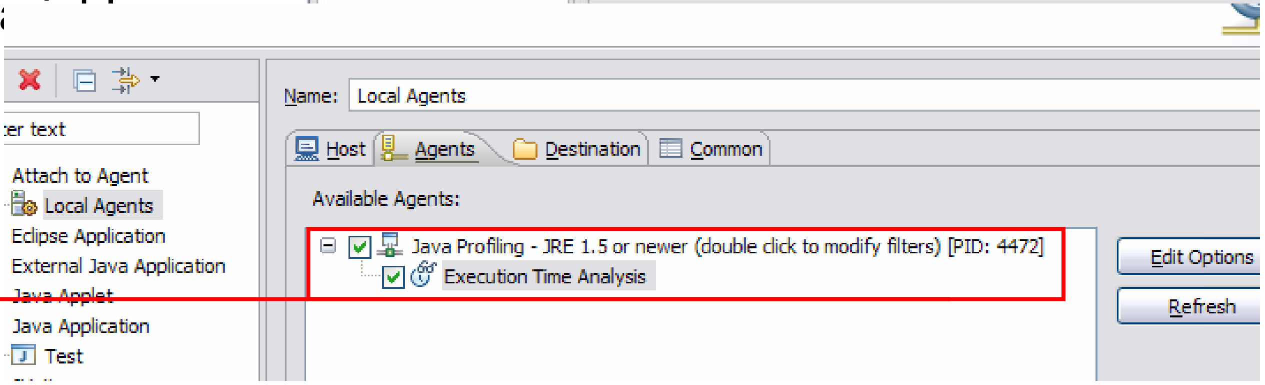
- Attach to Agent
  - Local Agents
- Eclipse Application
- External Java Application
- Java Applet
- Java Application
  - Test
- JUnit
- JUnit Plug-in Test
- Managed Resource
- OSGi Framework

Name: Local Agents

Host | Agents | Destination | Common

Default Hosts:

localhost[10002]
jgwl.torolab.ibm.com[10002]



Name: Local Agents

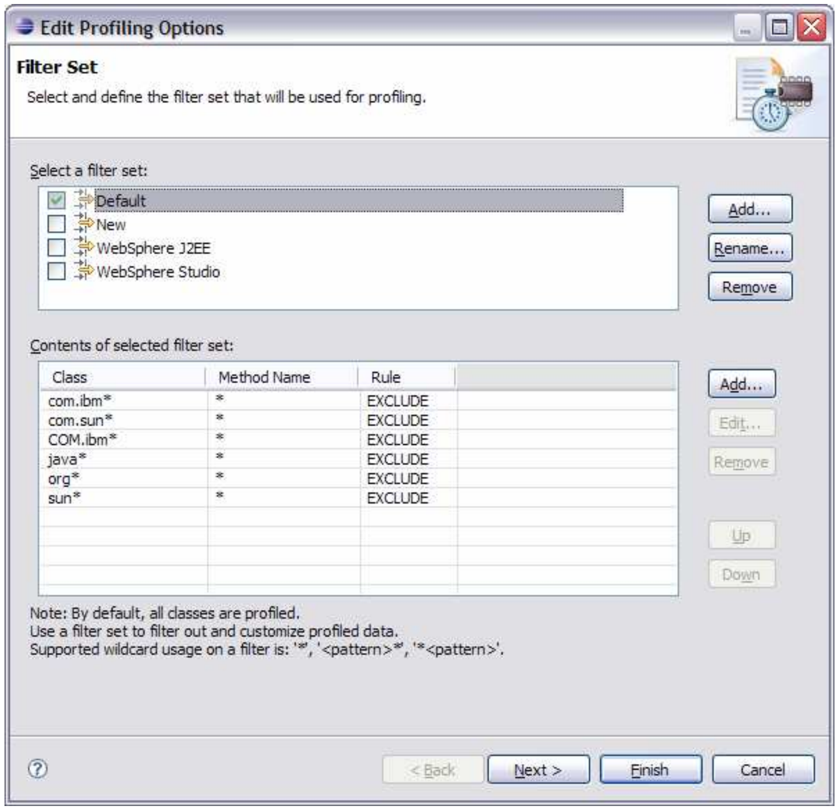
Host | Agents | Destination | Common

Available Agents:

- [✓] Java Profiling - JRE 1.5 or newer (double click to modify filters) [PID: 4472]
  - [✓] Execution Time Analysis

Filtering Options

- As previously discussed, profiling can slow down application execution significantly, if proper filters are not used.
- A default set of filters are available which will filter out a standard set of JVM classes.
- Filter options are:
  - Class (which can include the package name, as well as the class name.)
  - Method name
  - Inclusion / Exclusion



Filter Set

Select and define the filter set that will be used for profiling.

Select a filter set:

- [✓] Default
- [ ] New
- [ ] WebSphere J2EE
- [ ] WebSphere Studio

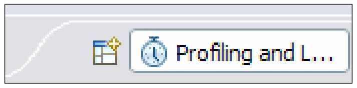
Contents of selected filter set:

Class	Method Name	Rule
com.ibm*	*	EXCLUDE
com.sun*	*	EXCLUDE
COM.ibm*	*	EXCLUDE
java*	*	EXCLUDE
org*	*	EXCLUDE
sun*	*	EXCLUDE

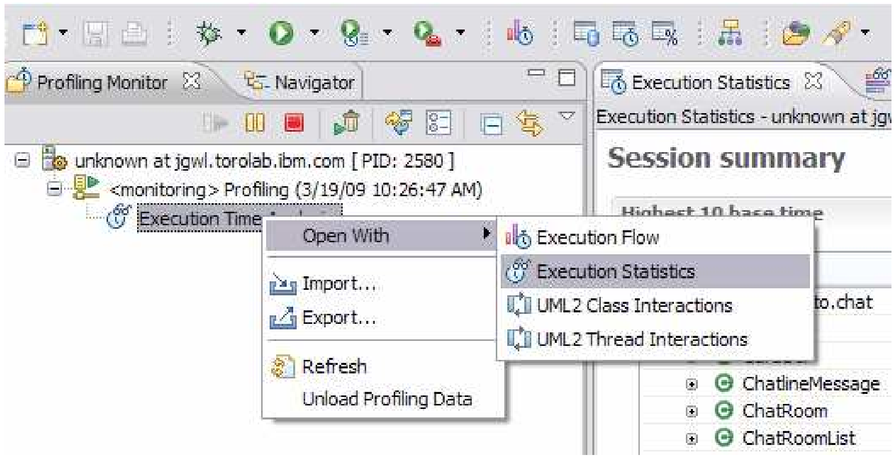
Note: By default, all classes are profiled.
Use a filter set to filter out and customize profiled data.
Supported wildcard usage on a filter is: '*', '<pattern>*', '*<pattern>'.

Profiling and Logging Perspective

- Once you have attached to your agent, you will be asked to switch to the Profiling and Logging perspective.

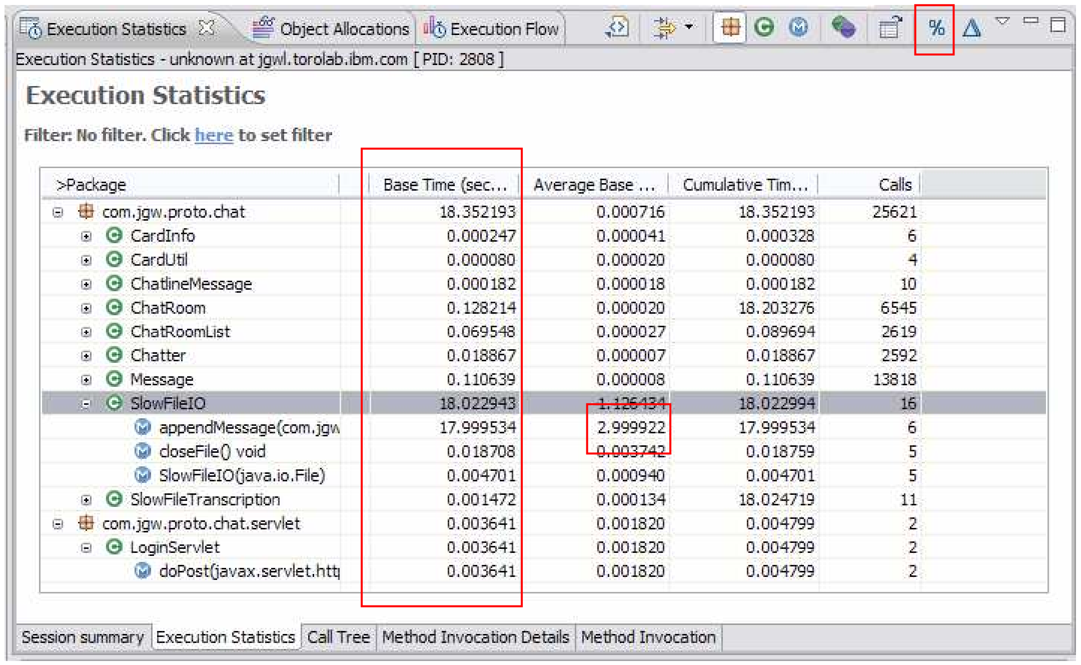


- The Profiling and Logging Perspective is the hub for all profiling functionality.



- Once you have attached to an agent, it will appear in the Profiling Monitor view, and profiling data will appear in their own views, based on the type of profiling you are performing.

Execution Statistics View



Execution Statistics - unknown at jgwl.torolab.ibm.com [PID: 2808]

Execution Statistics

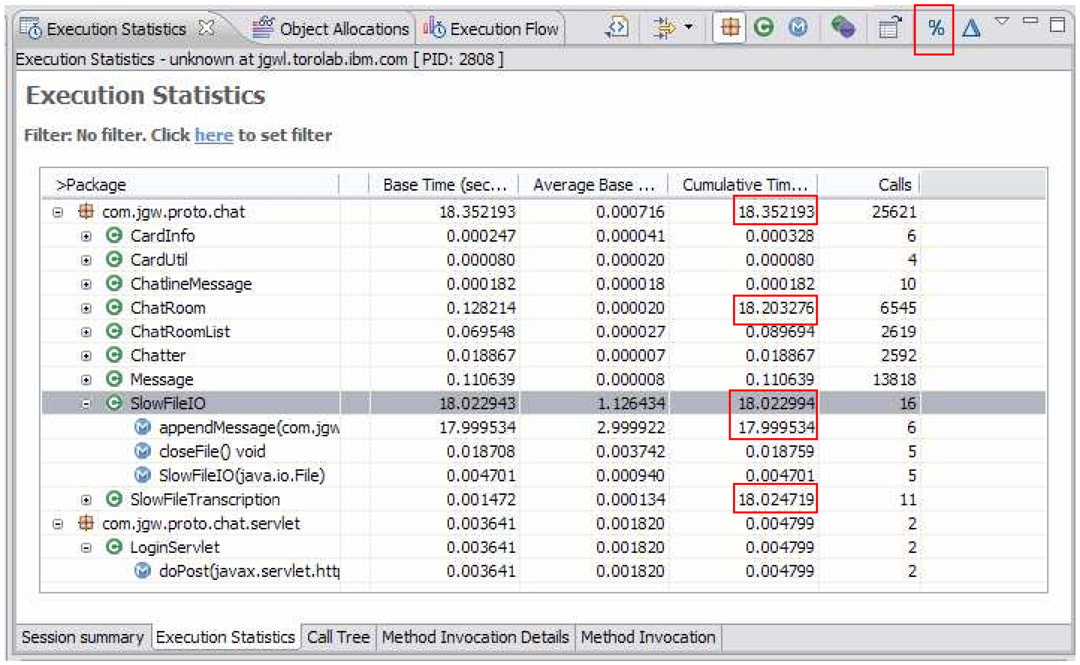
Filter: No filter. Click here to set filter

>Package	Base Time (sec...	Average Base ...	Cumulative Tim...	Calls
com.jgw.proto.chat	18.352193	0.000716	18.352193	25621
CardInfo	0.000247	0.000041	0.000328	6
CardUtil	0.000080	0.000020	0.000080	4
ChatlineMessage	0.000182	0.000018	0.000182	10
ChatRoom	0.128214	0.000020	18.203276	6545
ChatRoomList	0.069548	0.000027	0.089694	2619
Chatter	0.018867	0.000007	0.018867	2592
Message	0.110639	0.000008	0.110639	13818
SlowFileIO	18.022943	1.126434	18.022994	16
appendMessage(com.jgw	17.999534	2.999922	17.999534	6
closeFile() void	0.018708	0.003742	0.018759	5
SlowFileIO(java.io.File)	0.004701	0.000940	0.004701	5
SlowFileTranscription	0.001472	0.000134	18.024719	11
com.jgw.proto.chat.servlet	0.003641	0.001820	0.004799	2
LoginServlet	0.003641	0.001820	0.004799	2
doPost(javax.servlet.htt	0.003641	0.001820	0.004799	2

- **Base Time**: The time to execute the contents of the method itself, *excluding* calls to other methods. (In the chart, the Base Time field has summed together all calls of that method)
- **Average Base time**: The average time a particular method took to complete, excluding the time of method calls to other methods. (In the chart, this is the base time divided by the number of Calls)
- **Cumulative Time**: The time to execute the contents of the method itself, *including* calls to other methods.

- **Observations:**
  - Our JVM spent 18.35 seconds executing our application thus far.
  - Of that, 18.02 was spent in one particular package, and 17.999 was spent in one single method of that package.
  - On average, each invocation of that method took 2.9999 seconds to complete.
  - Using the % button at the top of the window, one can express the seconds as percentage of total, for convenience.

Execution Statistics View



Execution Statistics - unknown at jgwl.torolab.ibm.com [PID: 2808]

Execution Statistics

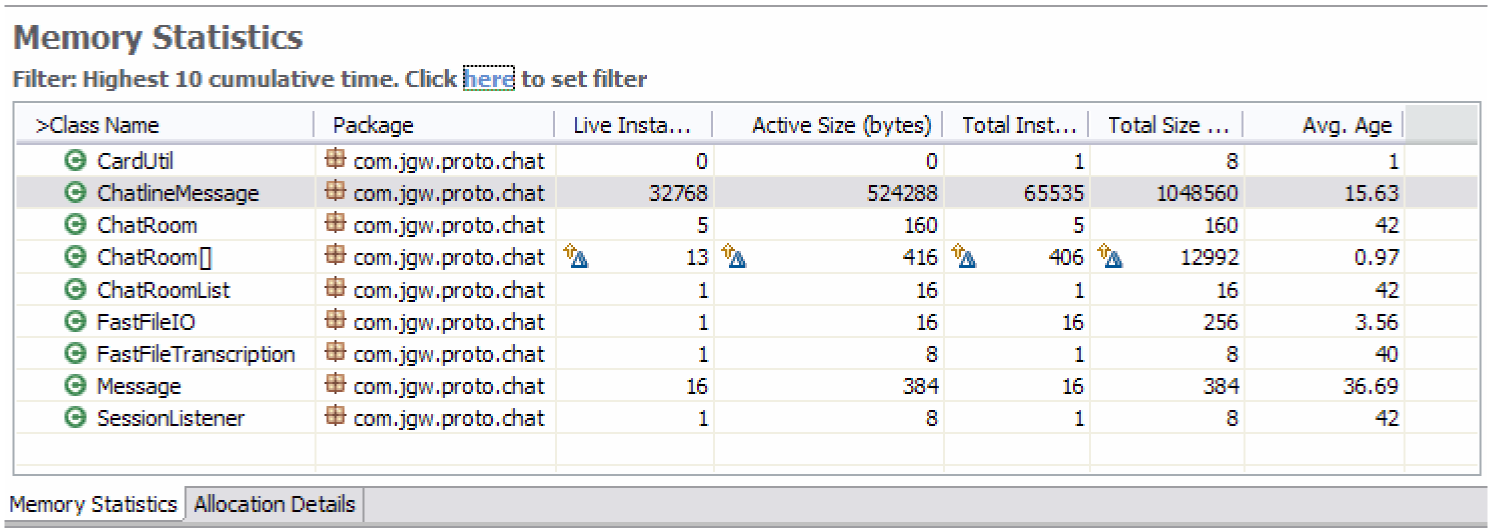
Filter: No filter. Click here to set filter

>Package	Base Time (sec...	Average Base ...	Cumulative Tim...	Calls
com.jgw.proto.chat	18.352193	0.000716	18.352193	25621
CardInfo	0.000247	0.000041	0.000328	6
CardUtil	0.000080	0.000020	0.000080	4
ChatlineMessage	0.000182	0.000018	0.000182	10
ChatRoom	0.128214	0.000020	18.203276	6545
ChatRoomList	0.069548	0.000027	0.089694	2619
Chatter	0.018867	0.000007	0.018867	2592
Message	0.110639	0.000008	0.110639	13818
SlowFileIO	18.022943	1.126434	18.022994	16
appendMessage(com.jgw	17.999534	2.999922	17.999534	6
closeFile() void	0.018708	0.003742	0.018759	5
SlowFileIO(java.io.File)	0.004701	0.000940	0.004701	5
SlowFileTranscription	0.001472	0.000134	18.024719	11
com.jgw.proto.chat.servlet	0.003641	0.001820	0.004799	2
LoginServlet	0.003641	0.001820	0.004799	2
doPost(javax.servlet.htt	0.003641	0.001820	0.004799	2

- **Cumulative Time**: The time to execute the contents of the method, *including* calls to other methods.

- **Observations:**
  - A slow child method (e.g. method at the top of the call stack) will be reflected in the cumulative times of its parent methods (calling methods).
  - Observing the packages and classes with a high cumulative time is one way to identify the call chain and determine what the offending methods are.

Memory Statistics View



Memory Statistics

Filter: Highest 10 cumulative time. Click here to set filter

>Class Name	Package	Live Insta...	Active Size (bytes)	Total Inst...	Total Size ...	Avg. Age
CardUtil	com.jgw.proto.chat	0	0	1	8	1
ChatlineMessage	com.jgw.proto.chat	32768	524288	65535	1048560	15.63
ChatRoom	com.jgw.proto.chat	5	160	5	160	42
ChatRoom[]	com.jgw.proto.chat	13	416	406	12992	0.97
ChatRoomList	com.jgw.proto.chat	1	16	1	16	42
FastFileIO	com.jgw.proto.chat	1	16	16	256	3.56
FastFileTranscription	com.jgw.proto.chat	1	8	1	8	40
Message	com.jgw.proto.chat	16	384	16	384	36.69
SessionListener	com.jgw.proto.chat	1	8	1	8	42

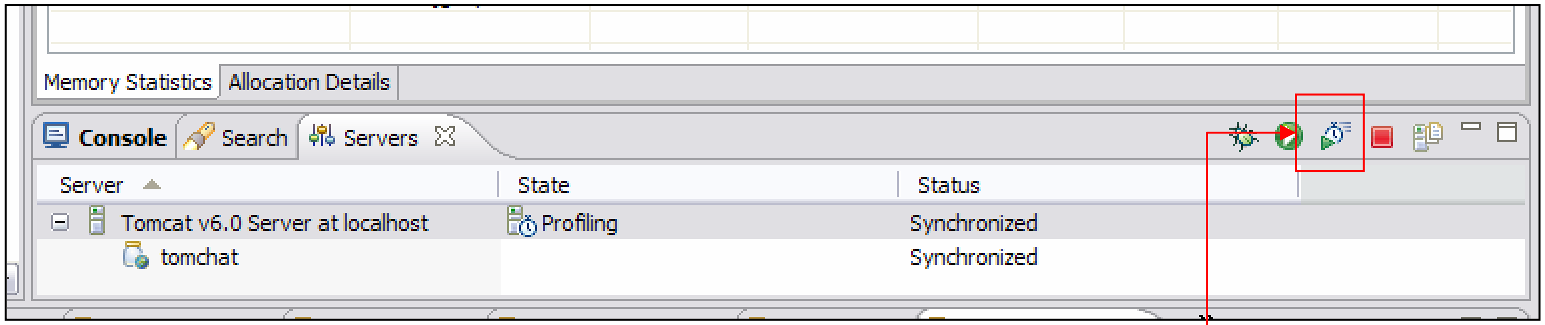
- **Live Instances:** The number of instances of the particular class that are still live in memory (have not been garbage collected.)
- **Active Size**: The total number of bytes in the heap that all live instances are presently consuming.
- **Total Instances**: The total number of instances of this class that have been created during the JVM's lifetime (including garbage collected objects).
- **Total Size**: The total size of all instances of this class that have been created during the JVM's lifetime (including garbage collected objects).
- **Average Age**: Average age of an object before it is garbage collected.

Profiling to File

- Instead of using a remote agent controller to profile a remote application, another option is to profile to file.
- In order to profile to file, use the following JVM option when starting Tomcat.
  - set JAVA_OPTS=-agentlib:JPIBootLoader=JPIAgent:server=**standalone**:file=**(output file name)**:filters**=(filters file name);<Profile-Option>**
- Where
  - **(output file name)** is the file that profiling data should be sent to
  - **(filters file name)** is a file describing the filters to use when profiling (this is highly recommended). See Getting Start document for details
  - **<Profile-Option>** as previously discussed.
- All profiling data will then be written to a file (the default file is trace.trcxml), which can be loaded from inside Eclipse under File -> Import -> Profiling file.

WTP Integration

- We have only touched on the topic of profiling locally during this presentation, but if you are launching your Tomcat server from inside Eclipse (rather than remotely) then you can access profiling capability straight from the Server view, without needing to set environment variables.



Server	State	Status
Tomcat v6.0 Server at localhost	Profiling	Synchronized
tomchat		Synchronized

Select the profile icon on a stopped server, and select the Profile on Server icon. A profiling dialog box will appear, and the server will start with the appropriate profiling options automatically.

- ➢Overview
- ➢Architecture
- ➢Agent Controller
- ➢TPTP Profiling
- ➢TPTP Testing
- ➢Q & A

TPTP Testing - Overview

- TPTP Test Tools provide a centralized, open-source, flexible and extensible framework for testing tools.
- Framework provides common tools for creating, managing and executing tests, deployments, datapools, execution histories and reports.
- Extends the TPTP Platform:
  - Common perspectives and views for interacting with target systems and resources.
  - Reference navigators, viewers, editors and wizards through extension points.
  - Standard EMF data model, query framework and assets repository.
  - Common data collection and execution framework on local and remote targets.
- Includes several test tool reference implementations used to test TPTP:
  - JUnit and JUnit Plug-in testing
  - URL testing

TPTP Testing - Model

- UML2 Testing Profile
  - Reference implementation of the UML2 Test Profile's standalone MetaObject Facility (MOF) model.
  - Definition model for the definition, creation and management of test artifacts including :
    - Test suites.
    - Test cases.
    - Datapools.
- Behavioral
  - Implementation of the UML2 Interactions meta model.
    - Loops
    - Invocations
    - Synchronization
- Execution History
  - Definition model for the definition, creation, management and persisting of test executions over time including:
    - Tests to be executed.
    - Deployments.
    - Locations.
    - Verdicts.
    - Attachments.
    - Messages and console output from the test execution.
    - User-defined custom attributes.
  - A collection of test execution traces and results, commonly referred to as a *test log.*

TPTP Testing - Concepts

- **Test Perspective**
  - Set of navigators, viewers, editors and wizards for tests and test assets.
- **Test Suite**
  - Consists of test cases and behaviors.
  - Created manually with the test suite editor or automatically by a recorder.
  - Hierarchical: can be contained by other suite
  - Executable with TPTP test execution framework.
  - Automatable Services Framework (ASF) : may be associated with a test script or Java class.
- **Datapool**
  - Provides input and expected output data to a test.
  - Consists of equivalence classes, variables and records.
  - Created with the datapool editor or imported from .CSV files.

TPTP Testing - Concepts

- **Test Execution**
  - Test assets deployment > execution by a test runner > loading results.
  - *Typically* started from Launch Configuration.
  - Automatable Services Framework (ASF) for launching tests programmatically via scripts (e.g. shell and ANT) and external applications (e.g. Java).
- **Test Log**
  - Persisted execution results.
  - Test log viewer to summarize, view, navigate and filter execution events, and associate defects.
- **Report**
  - Aggregates and summarizes numerous test execution results over a period of time (report window).
  - BIRT integration and extensible report generators.

TPTP Testing - Extensibility

- Extensible architecture to allow users to define vendor and product specific:
  - Test Creation Wizard
    - Extension point for defining wizard to create specific types of tests.
  - Test Editor
    - Extension point provided for associating editor with specific type of test.
  - Test Recorder
    - Extension point provided for registering a custom test recorder for a type of test.
  - Code Generator
    - Extension point for registering a custom code generator for a type of test.
  - Analyze Results
    - Extension point to open Log views used to analyze results of test run.
  - Publish Reports
    - Extension point for custom report types with custom content.
  - ASF
    - Extension API for custom execution.

TPTP Testing - Demo

- JUnit Testing
- URL Testing

Q & A

THANK YOU

TPTP on the web

- TPTP
  - http://www.eclipse.org/tptp/
    - Download
    - Documentation
    - Community: Newsgroup, mailing list
- TPTP Online Help
  - http://help.eclipse.org/
    - Monitoring and profiling applications
    - Testing applications

Legal Notice