Documentation of the Customer-based Service Monitoring (CSM) Tool

Manuel Günter
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10, CH-3012 Bern, Switzerland
`http://www.iam.unibe.ch/~mguenter`

Abstract

This document is a guideline for users of the CSM implementation [Gün01]. It shall help them to install and run the involved programs.

Contents

1 Overview

The CSM implementation is divided into three distinct programs that communicate over TCP sockets as depicted in figure 1.

- The *home application* allows the customer to send agents into the network. The program provides a graphical user interface that can also display the measurement and monitoring results that the agents send back. The application can also store these results on non-volatile media for analysis with other tools.

- The *CSM node* executes the customers' agents ensuring that no policy is violated. The program is run by the providers. The CSM node is the most complex part of the CSM implementation. It is connected to one or several border routers and aware of the neighbor providers' peer nodes. This is necessary for agent forwarding.
- The *T-component*. The CSM node gets the monitored UP packets form the T-component. The node tells the T-component what traffic its agents want to monitor and then gets matching IP packets encapsulated in a TCP connection.

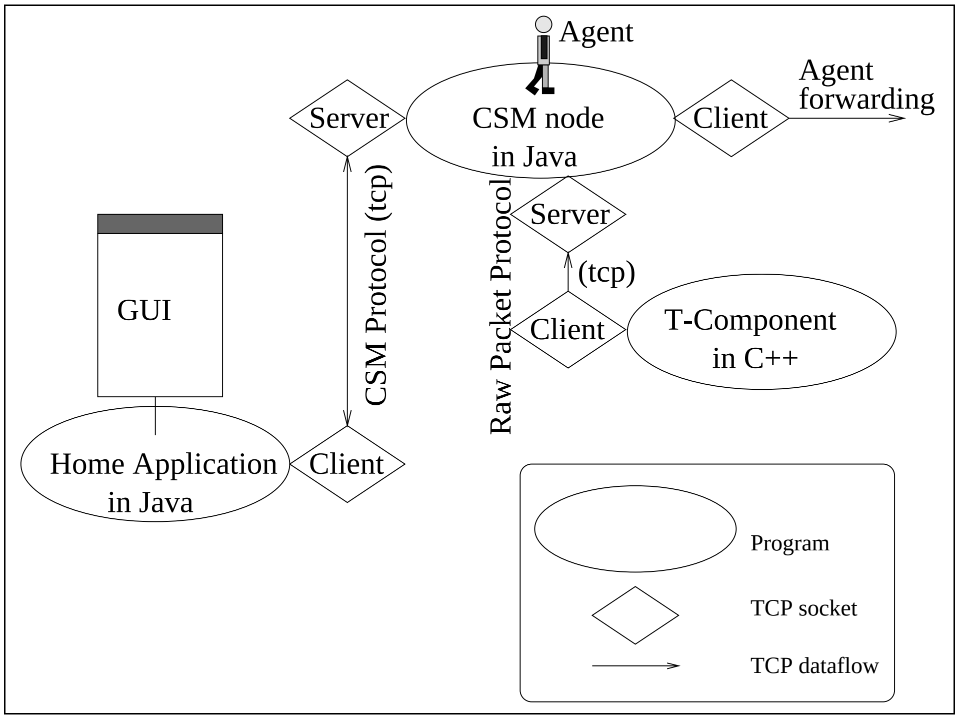

Figure 1: Implementation overview.

The CSM node and the home application are implemented in Java [Sun] (version 1.1.8). Then there are several kind of T-components: a Java dummy program, a script that starts Tcpdump [JLM89] and a C++ program, as well as a C++ program for virtual routers [BB00].

1.1 Organization of the Java Source Code

The CSM Java source code is grouped into packages. This provides more modularity, safety and managability to the implementation. The CSM implementation foresees two installations: one for the customer (the home application and individual agents) and the other for the provider (the nodes). Both installations have packages that are uniquely used by them, some that they both share, and some that are only stubs. The stub packages are necessary if one installation must know some basic classes of that package but not all of them. The best example is the `capsule` package that bundles the agents. For the node installation it suffices to know the `capsule.Agent` class. The individual agent class and its helper classes are dynamically downloaded by the CSM protocol when the customer sends the agents. Here is a complete list of the packages and their purpose:

- **`application.`** This package hosts stand alone Java applications that can be used in conjunction with CSM agents. An example application is a traffic generator.
- **`capsule.`** All CSM agents are implemented in this package.

- **`clientserver.`** The CSM protocol classes and helpers are implemented here.
- **`config.`** This package groups the classes that help the node or other applications read and parse configuration information from files.
- **`filter.`** All classes related to filtering are implemented in this package.
- **`homeApplication.`** The classes that implement the home application are bundled into this package.
- **`netgui.`** This package hosts the classes that help to display a network topology and callback agent results.
- **`node.`** This package bundles most of the classes relevant for the node implementation.
- **`topology.`** Here are the classes that implement the node routing.
- **`utils.`** This package contains helper classes that are also useful in other contexts such as e.g. the PGPEncoder class that provides access to PGP encryption and authentication.

Table 1 shows how the installations use the packages:

Table 1: The use of the packages by the two installation variants.

Package	Provider installation	Customer installation
`application`	stand-alone	not used
`capsule`	stub	used
`clientserver`	shared	shared
`config`	exclusively used	not used
`filter`	shared	shared
`homeApplication`	not used	exclusively used
`netgui`	not used	exclusively used
`node`	used	stub
`topology`	used	stub
`utils`	shared	shared

2 Installation

2.1 Software Prerequisites

- Java version 1.1.8.
- PGP 2.6.3i if encryption or authentication of agents is needed.
- A C++ compiler if the real T-component is needed.
- Shell script support (pipes and filters) if the real T-component is needed.
- The Tcpdump program, if the real T-component is needed.
- The virtual routers if they are needed.

2.2 Step-by-step Installation

2.2.1 The Provider Side (Server)

The Java Stuff.

1. Make sure you use Java 1.1.8. E.g. `module load java1`.
2. Go to or create a directory (e.g. */my/server/comes/here/*) where you would like to install the CSM node.
3. Decompress the file `provider.tgz` with `tar xvfz provider.tgz`.
4. Change the Java classpath to this directory. This operation depends on the operating system and shell you are using. E.g. `setenv CLASSPATH /my/server/comes/here/\:$CLASSPATH .`
5. You can start the CSM node now by calling: `java node.Node`. The output will probably complain about some fields in the default config file. It will also start a server for routing and later a server for the agents.
6. You can test if the server is running using telnet to the default ports 1997 and 1998. After sending a line the server will issue error messages, but it will not crash.

The T-component. The T-component is located at the provider site, but not necessarily in the same machine as the CSM node. If the T-component is a dummy then the Java installation will do fine. If the T-component is in a virtual router then you need to install virtual routers with so called t-bones. If the T-component is Tcpdump-based, then the CSM node interacts with C++ programs provided in the tar archive `t-component.tgz`. Unpack this archive in */my/server/comes/here/*`CPP/`. For more details see section 6.

2.2.2 The Customer Side (Client)

1. Make sure you use Java 1.1.8. E.g. `module load java1`.
2. Go to or create a directory (e.g. */my/client/comes/here/*) where you would like to install the CSM home application. Do *NOT* use the same path as for the CSM node as this would overwrite some classes that differ.
3. Decompress the file `customer.tgz` with `tar xvfz customer.tgz`.
4. Change the Java classpath to this path. This operation depends on the operating system and shell you are using. E.g. `setenv CLASSPATH /my/client/comes/here/\:$CLASSPATH .`
5. You now need to create a subdirectory called `topology`. This should be consistent with the topology directory in the provider installation. So either copy the contents of that directory or if possible, create a symbolic link to it:
   `ln -s /my/server/comes/here/topology /my/client/comes/here/topology`.
   The files in the topology subdirectory determine how your CSM nodes form an overlay network.
6. You can start the CSM home application now by calling: `java homeApplication.SDC`. (Note, that SDC is a 'historical' name and is synonymous to CSM). A little window will pop up that allows you to send queries or agents to CSM nodes.

3 Configuration

3.1 Configuration Files

3.1.1 The Node Configuration File

The CSM node is started with `java node.Node` *`configfilename`*. Usually, the node configuration files are located in the `config` subdirectory. The each line of a node configuration file contains one attribute-value pair. The pair is separated by semicolons. Note, that empty lines and line entries after a second semicolon are ignored. Here's an explanation of the attributes:

Attribute	Meaning
`NODE_ID`	Identifies the node. This is used in other configuration files and also by PGP. As a convention I use PGP-style E-mail addresses.
`PASSPHRASE`	The PGP passphrase to access the keys belonging to the node ID.
`NODE_LOOKUP_TABLE`	The full name of the file that holds the naming information. This table maps node IDs to IP and port numbers.
`NEIGHBOR_LOOKUP_TABLE`	The full name of the file that holds the topology information (for each node the table says what neighbor nodes there are). Note, that the last both tables are usually located in the `topology` subdirectory.
`PGP_TMP_PATH`	A place where PGP can create and remove temporary files.
`SCRIPT_PATH`	The path where the scripts for PGP and for the T-component are located.
`TEST_FILE_NAME`	The full name of a test file containing Tcpdump output. If this name is provided, then the node starts a T-component that reads from this file.
`T_IS_VIRTUAL`	If set to `true`, then the node assumes the presence of virtual routers.
`T_NAME`	This is the DNS name of the machine executing the T-component. If it is not set, then the T-component is local. Note, that if also no testfile is provided and `T_IS_VIRTUAL` is false, then the node starts a dummy T-component.
`T_DUMMY_INTERVAL`	The dummy T-component sends a packet every time after that much milliseconds.
`MAX_AGENTS`	That many agents may execute concurrently in this node.
`T_SERVER_PORT`	The C++ based T-components contact the node on this port up to this port plus MAX_AGENTS.
`Q_LEN`	The length of the packet queue that feeds the agent.
`CRITICAL_Q_LEN`	If the queue is filled to more than this value, then the emergency packet handling method is called.
`ENABLE_RESSOURCE_CONTROL`	If this is set to true, resource control (CPU and memory) is activated.
`CONTROL_INTERVAL`	The frequency of resource usage controls (ms).
`INTOLERABLE_LOSS`	If that many percent (between 0 an 1) packets are handled as emergency, then the node is considered congested.
`MIN_SIGNIFICANT_EXEC_LEN`	Do not consider agents that execute only for a short (ms) total time within an interval.
`MAX_TIME_2_NORMAL`	After a node congestion, wait that long (control intervals) before going back to normal operation.

Attribute	Meaning
MAX_MEM MAX_EXEC_TIME TIME_TO_LIVE MAX_BURST_LEN	The maximum number of bytes that an agent is allowed to use. The maximum total time (ms) that an agent may use the CPU. The maximum time (secs) that an agent may reside in the node. The maximum time (ms) per packet that an agent may cling to the CPU.
CONFIG_PATH USER_DB_FILENAME USER_DB_AS_OBJECT USER_DB_OBJ_EXT USER_DB_TXT_EXT	The location of the configuration files and the user database. The filename of the user database. The database can be represented as text or as a Java object. The file extensions depending on the file type.
MAX_USES_PER_SERVICE	If the service usage is not restricted otherwise, this is the maximum number of usages.

3.1.2 The Node Lookup File

The node lookup file is usually located in the `topology` subdirectory. As said before, the node configuration file provides the name and location of this file. The file structure is as follows (see also `topology/nodes.global.config` as an example): There is one line per node. Lines are separated in entries by colons (':'). The first entry gives the node ID. The second entry the DNS name of the machine hosting the node. The third entry (alternatively) provides the IP address. The forth entry gives the port that the node shall open for agent transmission. The fifth entry is the port that the node shall open for routing information. The sixth entry associates the node with an ISP organisation (a domain). Finally, there is a flag indicating if the ID really belongs to a CSM node (`false`) or if it is merely a customer (`true`). Customers may also be associated with and IP or DNS entry but they don't need the port numbers.

3.1.3 The Neighbor Lookup File

The neighbor lookup file is usually located in the `topology` subdirectory. As said before, the node configuration file provides the name and location of this file. The file structure is as follows (see also `topology/neighbor.global.config` - compare with figure 2): There is one line per node. Lines are separated in entries by colons (':'). The first entry gives the node ID. Further, entries give the IDs of nodes that are adjacent to this node (neighbors). Customers attached to a node are also in this list.

3.1.4 The User Database

The user database helps the node to classify users. It is usually located in the `config` subdirectory (see e.g. `userDB.txt`). There is one line per user. Lines are separated in entries by colons (':'). The first entry gives the user ID (as used in the node lookup file and by PGP). The second entry declares what the home networks of that user are. Currently, there may be any number of space separated subnet addresses. The addresses are specified in the decimal form (e.g. 130.92.64.4). If digits are missing they are considered as wildcards, thus 130.92 would be equivalent to 130.92.0.0/16. The third field specifies what kind of policy this user is going to expect. It thus says what type of customer this is. The forth field is optional. It contains a space separated list of privileged service numbers to which that customer has access.

Policies. The policy object describes e.g. if the user must authenticate or if it is a super user and the priority of his/her agent. Most importantly, it holds the filter that regulates which traffic this user's agent is able to see (see `node/Policy.java`). The policy object for an agent is created during runtime by a policy generator (subclass of `node/PolicyGenerator.java`). A policy generator represents a policy group (a user type; thus the user database contains for each user the name of a policy generator). There are the folowing policy generators (feel free to implement new ones):

- `node/SuperPolicyGen.java`. For super users.
- `node/CustomerPolicyGen.java`. For regular customers.

- `node/AnonymousUserPolicyGen.java`. For customers that want to stay anonymous (do not authenticate). These kind of users have restricted rights (e.g. in filtering).
- `node/TestPolicyGen.java`. Like customer the policy generator but also without mandatory authentication.

3.2 In-Code Configuration

Not all configuration options are available in the configuration files. The home application for example does not have a configuration file. Usually, the user can provide the information in the GUI form fields. Some values, however are within the Java byte-code: You probably need to adapt the values and recompile these files.

- In `homeApplication/ExecutionMessageForm.java`: Change `path` to the home directory of your client side application.
- In `homeApplication/SDC.java`: Set `topoPath` to where the topology files are.
- In `homeApplication/CallBackDisplay.java`: set `savePath` to where you would like to save callback results.
- `netgui/EndUser.java`: adapt the path in the `loadImage()` method call.
- `netgui/Router.java`: adapt the variable .
- `netgui/.java`: adapt the variable `imgName` and `imgSelName`.

The configuration of the T-component is also not file based. The Tcpdump-based T-component is started by scripts that are located in `utils/Scripts/`. Usually, these scripts are one-liners that take most of the parameters as arguments. This also holds true for the PGP scripts which are also located there. Support of other PGP versions or encryption tools is also possible but then the appropriate Java classes must be adapted. For that purpose have a look at utils/PGPEncoder263i.java.

For more details about where to configure what, see also the file `config/README`.

4 Agents

The agents' code is (obviously) stored at the customer site, in the subdirectory
`/my/client/comes/here/``capsule` (the directory name is historical and comes from active networking).

New agents should go into this directory and should be declared to belong to the capsule package. Before an agent can be used, it must be compiled. Further, a *`agentname`*`.names` file must be created that contains all superclass names and all helper class names. This is necessary so that these classes can be transmitted with the agent.

Note: there's two kinds of agents: the ones that do not use *callback* and the ones that do. If agents use the forwarding mechanism, then they cannot use the initial TCP connection for transmitting results. Instead they must use the call back service of the node.

Here is brief description of each agent in this directory (alphabetical order):

```
Name:   ActiveBWbottleneckAgent
Goal:   Compares the consecutive arrival times of packets that match
        a hash signature in order to derive the bottleneck bandwidth.
CallBack:   no (but there is commented out code to turn it into one).
Abstract:   no
Extends:   HashAgent

Name:   Agent
```

```
Goal:   Interface description
CallBack:   no
Abstract:   yes
Extends:   -

Name:   BandwidthAgent
Goal:   Calculates current bandwidth usage on an interval basis.
CallBack:   no
Abstract:   no
Extends:   MeterAgent

Name:   ConsistencyAgent
Goal:   Checks if the IP packets are consistent:
        e.g timestamp order, length, checksum.
CallBack:   no
Abstract:   no
Extends:   MeterAgent

Name:   DebugAgent
Goal:   Sends back every packet copy it receives.
CallBack:   no
Abstract:   no
Extends:   MyAgent

Name:   DoSAgent
Goal:   To demonstrate a denial-of-service attack against the node.
        Various variants are there (in commented form).
CallBack:   no
Abstract:   no
Extends:   -

Name:   FastestAgent
Goal:   A minimal agent that does nothing. Used for
        performance tests.
CallBack:   no
Abstract:   no
Extends:   -

Name:   FastSendBackAgent
Goal:   As soon as started, this agent sends a single result back.
        This is to test the agents 'rtt'.
CallBack:   no
Abstract:   no
Extends:   MyAgent

Name:   GrowerAgent
Goal:   For each received pckt this agent stores an integer. It thus
        grows infinitely (to test memory consumption).
CallBack:   no
Abstract:   no
Extends:   -

Name:   HashAgent
Goal:   This agent may hash the packet and compare to given bits (1-64).
```

```
        Agents that want to react to specific packets (e.g. generated
        by a tool - see application/Trigger.java) inherit from HashAgent.
CallBack:   no
Abstract:   yes
Extends:   MyAgent

Name:   KillAgent
Goal:   Requests a wakeup call to kill something after it
        has been initialized.
CallBack: no
Abstract: yes
Extends: MyAgent

Name:   KillAgentsAgent
Goal:   Kills all running agents in a node.
CallBack:   no (but can be forwarded)
Abstract:   no
Extends:   KillAgent

Name:   KillNodeAgent
Goal:   Terminates the node.
CallBack:   no (but can be forwarded)
Abstract:   no
Extends:   KillAgent

Name:   MatchCounterAgent
Goal:   Counts the number of packets matching a hash.
        Testing and tutorial example.
CallBack:   no
Abstract:   no
Extends:   HashAgent

Name:   MeterAgent
Goal:   This agent defines the interface for agents towards meters.
        Meters are accumulators that transform a stream of packets into
        an array of results. From the result array the meters also
        calculate an accumulation (e.g. the average).
        This directory contains a large number of meters that collaborate
        with agents (*Meter.java).
CallBack:   no
Abstract:   yes
Extends:   MyAgent

Name:   MultiMeterAgent
Goal:   Agents that use several meters inherit from this class.
CallBack:   no
Abstract:   yes
Extends:   MeterAgent

Name:   MyAgent
Goal:   The superclass of basically all agents. Implements some
        helper methods for initalization and to call node services.
CallBack:   no
Abstract:   yes
```

```
Extends:   -

Name:   OneWayDelayAgent
Goal:   Registers the arrival time of packets that match a signature.
        With synced clocks the results of several OneWayDelayAgents can
        be compared to derive jitter and one-way delay.
CallBack:   yes
Abstract:   no
Extends:   HashAgent

Name:   PassiveBWbottleneckAgent
Goal:   Compares the size and arrival time of (any) packets to derive
        the local bottleneck bandwidth.
CallBack:   no
Abstract:   no
Extends:   MyAgent

Name:   PeekAgent
Goal:   Same as the DebugAgent.
CallBack:   no
Abstract:   no
Extends:   MyAgent

Name:   PerformanceAgent
Goal:   Uses a BandwidthMeter to accumulate x packets and send their
        bandwidth consumption.
CallBack:   no
Abstract:   no
Extends:   MeterAgent

Name:   PerformanceAgentD
Goal:   Like PerformanceAgent, but opens an own connection (forwarding).
CallBack:   yes
Abstract:   no
Extends:   MeterAgent

Name:   PingListenerAgent
Goal:   Monitors pings (icmp rfc 792). Matches requests to replies,
        accounts arrival times. Derives packet anomalies
        (reordering, loss, duplication).
        Calculates jitter on the assumptions that pings were sent regularly.
CallBack:   yes
Abstract:   no
Extends:   MyAgent

Name:   TestAgent
Goal:   Like DebugAgent. Contains also a number of (commented) statements
        to trigger a security exception.
CallBack:   yes
Abstract:   no
Extends:   -

Name:   TriggerAgent
Goal:   Wraps any other agent and starts it once a packet (trigger) matches
```

```
        a given hash.
CallBack:   no
Abstract:   no
Extends:   HashAgent

Name:   VPNAgent
Goal:   Validates the encryption of VPN packets and also measures the bandwidth
        consumption on an interval basis (using the appropriate meters).
CallBack:   no
Abstract:   no
Extends:   MultiMeterAgent
```

5 Use Cases

In this section we use some of the above agents in order to learn about the functionality of CSM. Just follow these tutorial-style steps to encounter implemented CSM functionalities.

5.1 Preparation

- Install CSM as described in section 2 (without starting the programs).
- Go to the topology subdirectory and create a neighbor lookup file and a name lookup file, or adapt an existing one. The format of these files is described in section 3.1. Note, that when you want to use PGP, then the user and node IDs must exist in your PGP key ring (simply create new names and keys for these IDs - try `pgp -h`).
- Adapt the `config/default` file so that it uses your topology files. Adapt the `NODE_ID` and the `PASSPHRASE` attribute (so that they match your PGP key ring).

5.2 Demo I: One Node, One Agent

In this scenario we run one node and monitor test traffic that comes from a file of from a dummy T-component with a couple of simple agents.

- Start the CSM node (`java node.Node`).
- Start (in a separate shell) the home application (`java homeApplication.SDC`) and perform the following things:
  - Press on the query button. You will get a form. Enter the receiver ID that corresponds to the node that you just have started.
  - In case you changed the user database, you may also need to enter another sender ID.
  - The encoding field lets you select two cryptographic options or plain text.
  - Selecting the crypto stuff helps you check if your PGP installation was done right.
  - There is only point-to-point forwarding for queries so you basically can't select another forwarding option.
  - There are six query types. These are described in chapter 5.1.5 of my thesis [Gün01].
  - When you send a query with the send button, there should be a response window popping up immediately.
- Go back to the main window of the home application.
- Press the send button to get to a form that allows you the transmission of agents.

- You see that the top fields of the form are the same as for the query.
- Select the agent: `capsule.PeekAgent`. You can do so by typing or by browsing the file system.
- This agent does not use callbacks, so don't press that switch.
- Then there are a couple of fields that allow you to compose a filter to be used by the agent. Note the wildcard character '`*`' (not working for IP addresses, though). IP addresses can be entered the same way as in the node lookup file. Note, that for all filter fields but the last two you can enter several numbers (separated by blanks). The packet length field trims the original sniffed packets to that length (e.g. if you just want to get packet headers). The 'number of packet' field lets you select the number of packets that the agent wants to examine. For the given agent set this field to `1`. After you have entered something, don't forget to press <return> (else your entry is not used). If you entered rubbish, the field will show it. Note, that the filter is not used when sniffing from a T-component dummy or a test file (only when sniffing with Tcpdump or with the t-bone). You can easily turn filtering on by uncommenting the appropriate line in the `run()` method of class `node.AgentWrapper.java`. But then be aware that the generated test traffic (or the one on file) will probably not match the filter and your agents won't see a thing.
- Send the selected Peek agent by pressing the send button.
- All this agent does is to send a copy of the monitored packet backs to the home application. A window will pop up and display the packet. You can display the packet in different graphical and textual styles. Try it!
- Now select a `capsule.BandwidthAgent`. This one measures the throughput on an interval basis.
- Set the number of packets to 100 packets.
- Send the agent and watch the result. This is the dummy traffic that is being monitored. If you want to change the dummy traffic source, have a look at `node/T_ComponentDummy.java` in the server installation. A simple way to increase the traffic is by changing the `T_DUMMY_INTERVAL` attribute in the node configuration file.

Now we have a look at the VPN agent which measures the throughput on a VPN tunnel and performs a crypto check. It will immediately balk when the dummy traffic is used, because it is not encrypted. We will now use a T-component that reads from a test file.

- Adapt the `config/default` file so, that the testfile `ahEsp3000x1024.tcpdump.output` is used. Inspect this file and learn that it is a Tcpdump output of IPSec tunnel traffic.
- (Re-)Start the node.
- Send the `capsule.VPNAgent` to the node.
- The home application will display the agent's results. The agent measures the throughput of the packets (not the throughput now, but the one that was seen when the testfile was generated). Some packets (it should be 2 percent of them) will fail the crypto test. When that happens the agent sends the packet back and terminates.

5.3 Demo II: Security

Use the dummy T-component or a test file for these tests.

- Try to send an agent with the sender ID set to a customer (a user with `CustomerPolicyGen` in its user database entry). The node will complain that you must authenticate.
- Thus, select one of the cryptographically protected encoding schemes when sending the agent.

- Now, have a look at the agent `capsule/DoSAgent.java`. There are a number of (commented) denial-of-service attacks implemented in that agent.
- Uncomment what you like and send the agent to see the resource control.
- The `capsule/GrowerAgent.java` gradually increases its memory consumption. Observe how the resource controller will terminate it after a while.

5.4 Demo III: Distribution, Forwarding and Callbacks

In this scenario we use T-component dummies. We will set up a scenario with several nodes at the same time. Thus, you need to plan on which machine to run which node. You have to fill in this information into the node lookup file (see section 3.1.2) and into the neighbor lookup file (see section 3.1.3). It is possible to run more than one node on one machine, but you must carefully avoid that a port number is used more than once. Two of the ports are declared in the node lookup file, but the ports for their T-components are declared in the node configuration files (the `T_SERVER_PORT` attribute). Be aware, that the latter attribute only describe the lowest number of a *range* of ports, going up to `T_SERVER_PORT+MAX_AGENTS`.

In the provided topology subdirectory you will find the files `nodes.global.config` and `neighbor.global.config` which can help you as an example. They define the scenario as depicted in figure 2. Each of the four nodes use their own configuration file provided in the `config` subdirectory. We use the Unix shell script `utils/Scripts/multipleNodeScenario` to start the nodes in the appropriate machines. The script uses yet another script called `startAnode`. Note, how this script starts the nodes with an argument. The argument composes of the DNS name and the port to be used, but it is mainly used to point to the node configuration file that this particular node must use. Make sure to adapt these shell scripts (especially the paths they use). Also, the node configuration files should be in the place and have the name that the script expects them to: `*/my/server/comes/here/*config/*DNSname:port*`. Make sure the paths of these configuration files are valid and adapt these files to your likings. Note, that the output of the nodes is redirected to files in the `Logs/` directory. Make sure this directory exists and is empty.

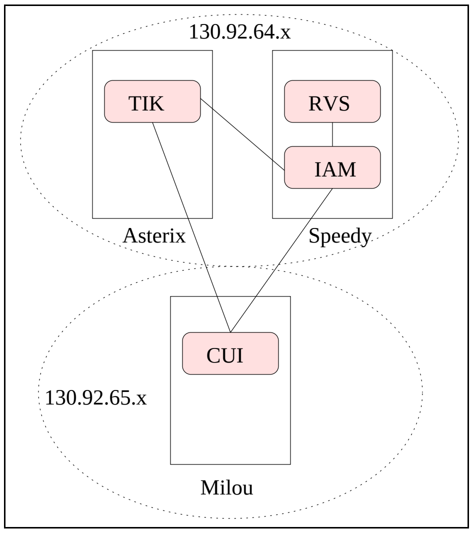

Figure 2: The multi-node scenario.

Bhew, now we are more or less ready to go. If you have forgotten some paths, then the following steps will reveal that for sure.

- Start the nodes with the script.
- Start the home application.

- Test if the remote nodes are alive by querying them (enter the desired receiver node ID in the form). Topology problems can be revealed by query the routing information.
- If something goes wrong, make sure you stop all nodes before retrying it.
- Open the agent sending dialog.
- Chose a sender ID. The user should be at least a customer (compare with the user database), so it can authenticate (which is mandatory for forwarding).
- Select an encoding option (`SIGN_ ONLY` or `PGP` - both authenticate the sender).
- Set forwarding to `Broadcast`.
- Select the `capsule.PerformanceAgentD`. This is a throughput measuring agent with callback capabilities. Callback is necessary, because we are going to use forwarding.
- Select the *start callback server* switch. The GUI is then going to ask you what is measured and in what range the measured values will be (for convenient display). Select Min=0, Max=1.5.
- Send the agent. Fill in the passphrase when asked and observe the new popup window.
- As the agent is forwarded the home application learns about the network topology and displays it.
- You can rearrange the display location of the routers and of the end user hosts by drag-and-dropping with the mouse.
- If the *record* switch is on (default), then the home application stores all results. To save the results to disk you have to select a router of interest and press on the *save* button. The results are saved on a file as indicated in the text field. The resulting file is formatted in gnuplot style.
- Try now sending an agent with `Hop-by-Hop` forwarding. There, you need to specify an IP address of a customer as a target. The agent will first be sent to the receiver node. From there on it will be forwarded to each node on the path towards the target IP. To get a valid target IP inspect your node lookup file (there, also the user IDs and IPs are listed). With the network of figure 2 try for example to send the agent to `<admin@cui.unige.ch>` and have it hop-by-hop forwarded towards 130.92.64.4 (a user that is connected to the node `<admin@tik.ethz.ch>`.
- A convenient way to stop the distributed nodes is to broadcast a `capsule/KillNode.java` agent. This agent uses a privileged service, so you need to authenticate as a super-user.

6 Live/Real Monitoring

Up to now, we have not performed any live monitoring. Live monitoring involves T-components at strategic places that have the access rights to sniff traffic. These components are completely outside of the Java implementation parts of CSM.

6.1 The Tcpdump T-Component

This component supports live sniffing on a real network. Here is how you can set up such a T-component.

- Plug a Linux machine to a network to be sniffed. The machine should have Tcpdump installed.
- Install the T-component into the directory of your choice. The archive `t-component.tgz` contains two subarchives:
  - `linux-fast.tgz`. This is the version you will probably prefer. It uses the raw output of tcpdump and is thus faster than the second choice, which uses the human readable output:

  – `linux-HumanReadable.tgz`. Note, that to use this one here, you need the `Sender.o` of the first archive.

- Recompile the T-component if needed.
- Adapt the script `tComponent` (a UNIX shell script that pipes the Tcpdump output into a C++ program that parses the output and sends it to the CSM node).
- Go back to the Java provider installation site.
- Adapt the node configuration file: Provide the DNS of the machine running the T-component in `T_NAME`. Make sure, that no testfile is provided and that the virtual router attribute is set to false.
- Adapt the paths of the script `utils/Scripts/start_T_remote_live`
- Start the node and try to send a `capsule/PeekAgent.java`
- Don't forget that (1) You need to generate traffic that (2) mtaches the filter of your agent and (3) matches the filter policy of the owner of the agent (see also my thesis section 4.3.3).

6.2 Sniffing on Virtual Routers

First you need to establish a network with virtual routers (VR). On each virtual router the `t-bone` program should be running. This document does not describe how to configure VRs. We refer to the author Florian Baumgartner (`baumgart@iam.unibe.ch`). A little information can be found in `Doku/vrQuickHowTo.txt`.

To tell a CSM node that it should monitor a VR you need to set the `T_IS_VIRTUAL` attribute to true. You also have to set the `T_NAME` attribute to the DNS name of the machine that hosts the virtual router (not to the IP of the VR itself!). The CSM node will contact the `t-bone` on a port which is given by: The node's server port (the forth entry in the node lookup table) plus the `clientserver.ProtoConsts.VR_PORT_OFFSET` (set to 1000). Thus, when you start the t-bone in a VR you need to consider this portnumber `t-bone -p` *`portnr`* (typically 1998+1000=2998).

References

[BB00] Florian Baumgartner and Torsten Braun. Virtual routers: A novel approach for qos performance evaluation. In *QofIS'2000*, September 2000.

[Gün01] Manuel Günter. *Managment of Multi-Provider Internet Services with Software Agents*. PhD thesis, University of Berne, June 2001.

[JLM89] V. Jacobson, C. Leres, and S. McCanne. Tcpdump. available via ftp to: ftp.ee.lbl.gov, June 1989.

[Sun] Sun Microsystems. The source for java technology. http://java.sun.com/.