Extending snBench to Support a Video-Based Intrusion Detection and Alerting System with a Centralized Hash Table ‡

Dustin Burke  Dave Cecere  Ben Freiberg
dburke@cs.bu.edu  dcecre@cs.bu.edu  bfrocks@cs.bu.edu

Computer Science Department
Boston University
September 16, 2006

‡ This report is derived from the documentation of a final project submission for a software engineering class at Boston University. The authors have since graduated and, despite some layout adjustments, the content of this report is the same as that of the original submission. This report has been published so that the work described within may be referenced in future reports and papers. —Michael Ocean

Abstract

In this project we design and implement a centralized hashing table in the snBench sensor network environment. We discuss the feasibility of this approach and compare and contrast with the distributed hashing architecture, with particular discussion regarding the conditions under which a centralized architecture makes sense.

There are numerous computational tasks that require persistence of data in a sensor network environment. To help motivate the need for data storage in snBench we demonstrate a practical application of the technology whereby a video camera can monitor a room to detect the presence of a person and send an alert to the appropriate authorities.

1 Introduction and Motivation

This report presents the details of our project submitted in partial fulfillment of the requirements of the course. Included is documentation pertinent to developers and researchers of the snBench initiative[1], involving design specifications as well as the operation and maintenance of our module. Project internals are discussed (team roles, development schedule, etc.) as well as our reflection on our semester progress. We conclude with a thorough analysis of the benefits, costs and limitations of our approach with suggestions for future enhancements.

Our project module is the end product of the following problem statement: “To expand opcodes to enable STEP programmers with the ability to store and retrieve snObjects, both for historical analysis as well as system-wide parameters in the Sensorium.”

2 Components and Features of Project

Our module consists primarily of centralized hashing opcodes and the classes necessary to support a client / server architecture. As auxiliary to our main aim we have also implemented an extensive math library, a small feature set of image processing opcodes, and an email opcode supporting outgoing messages via an smtp server.

Centralized Hashing Opcodes

This module involved the creation of a centralized storage model that allows STEP programmers to transparently perform “put” and “get” operations without concerns for where the data is physically housed or how it is retrieved in the sensor network. We’ll elaborate on the specifics of the design necessary to support this client / server model of storage in sections below.

For STEP programmers, the API they use to leverage this module is as follows:
snBoolean sxe.core.hashput(String key, snObject value);
snObject sxe.core.hashget(String key);

Although the above functional specification isn't in the syntax of STEP its semantics are clear. The syntax a
STEP programmer would use is illustrated by the following example STEP program fragment:

<exp id="hashput01" opcode="sxe.core.hash.put">
  <value id="hashkey01">
    <snobject type="snbench/string">testKey</snobject>
  </value>
  <value id="hashval01">
    <snobject type="snbench/string">This is the string to store.</snobject>
  </value>
</exp>

Math Opcode Library

The emphasis on the math opcode library was in supplying the most commonly used mathematical functions as well
as those predicted to have most utility for STEP graph computations. Existing arithmetic functions (add, subtract,
multiply, divide) were augmented and overloaded to support arguments either of type sninteger or of type snDouble,
a new type we created in response to demand. The opcodes are overloaded in the sense that both operands must not
be of the same type but will correctly be coerced to the correct type to avoid incorrect arithmetic operations due to
truncation, etc. Among the functions supported, a short list includes arithmetic functions, trigonometric functions,
radian and degree unit conversions, absolute value, ceiling, floor, min, max, modulo, exponentiation and logarithmic
function.

Email Opcode

The email opcode was created in response to demand. This opcode provides STEP programmers with an alternative
reporting method via outbound-only email messages. The opcode makes use of a java mail framework that provides
an on-demand, embeddable smtp server to send out the message.

Image Processing Opcode Library

We have implemented several basic image processing functions that operate on snImages. Specifically, the opcodes
include image differencing, motion detection, gray-scale, thumbnail, load image, and save image. The motion de-
tection is very basic and simply performs an image difference between two consecutive frames and if the number of
pixels that have changed in intensity value exceed some threshold then report that motion has occurred. We discuss
some limitations of this approach as well as proposed extensions in later sections.

3 Team Roles

Our roles converged on all aspects of analysis and design concerns. Work tasks were divided with two primary
motives: (1) to match problems with the team member most capable of finding its resolution and (2) to decouple,
modularize and render semi-independent the sections of code that we each coded. The reason for the first aim is
self-evident but had a direct impact upon the timely completion of work items, especially under time constraints
(the primary risk involved in the project). The desire to modularize the code is natural to object-oriented paradigms
such as encountered in the snBench code-base; however, this modularization also reduced the overhead necessary
with merging code completed in parallel by numerous programmers. We made the executive decision to not use
source control and instead managed to coordinate development in this modularized fashion. Dave primary handled
the development of the intercommunication between client and server processes as well as development of the hash
table. He re-factored several classes of code to adhere to accepted design patterns. Cache invalidation and update
mechanisms were developed in conjunction with Dustin. Dustin also designed the garbage collection component and
numerous imaging processing opcodes. Ben did most of the heavy lifting when it came to opcode development. His
primary focus was with supplying an extensive math library for STEP programmers.
4 User’s Guide

To activate and use our package, first download the source zip-file and extract. A folder will be created that acts as the root of the project when importing into Eclipse. In Eclipse (the recommended IDE) simply create a new project from existing source and choose the snBench folder as the root. If you are merging our modules with an existing snBench code-base, please refer to the file “CHANGES.TXT” that enumerates which dependent files have been modified. As described in the section above “Organization of the Java Package,” you will find the majority of the module files contained in the sxe.core.hash package. If you desire to use the sxe.core.email package, you will need several framework jar files. Details on how to acquire and install these libraries are contained in the file “REQUIRES.TXT.” It is highly recommended that you also install the Java Media Framework (jmf.jar) to take advantage of the frame grabbing capabilities of snBench. Again, I refer the interested user to “REQUIRES.TXT” for details. To use the module, the STEP programmer is referred to the API specifications given in sections above regarding the syntax of the opecodes we provide. A compelling example of the syntax can be found in the file “sdt.xml.” To properly run this demo, we refer you to the file “DEMO.TXT” which details the necessary steps to run. Developers, refer to our website under the section Developer’s Notes where you can find the JavaDoc detailing the API specifications of all class files involved. I will now walk you through a quick demo of our product. Open a terminal window, change to the root directory of the snBench code and execute:

```java
java sxe.Server http://localhost:8080 NONE
```

This will fire up the SXE Server. Open a new terminal window and execute:

```java
java sxe.core.hash.CentralizedHashStorageServer 1212 i
```

The central resource will now create a listening socket. If you desire this process can be run from a different machine. To do so, make sure to specify the ip address of the central resource in the file “storageenetwork.xml” along with the matching port number on which the server binds and listens. Next, open a third terminal window and type:

```java
java sxePoster http://localhost:8080
```

This last command will bring up an interactive dialog that allows one to “post” a STEP file to the executing SXE server. From the menu, select option [1]. From the enumerated list of STEP files, select the number corresponding to “setThresh5.xml”. You may wish to read the console output of the SXE server to see traces of the execution. Next post the file “reportTempOver.xml”. This STEP file will write out to the console an “Alert!” if the temperature detected by a thermometer exceeds the specified threshold (verify that it does). Next post the file “setThresh300.xml” and notice what happens. The SXE Server should stop alerting that the temperature threshold was exceeded. To see the local cached copy of keys and values, open up a web browser and point to the url:

http://localhost:8080/snbench/sxe/storage/

The trailing slash is required. You can get and put hash entries via this command interface. Monitor the SXE Server’s console window to see the messages. To kill the SXE Server and CentralizedHashStorageServer, issue the kill command (CTRL-C). The SXE Poster can be quit using the “quit” command. Close the terminal windows.

5 Testing Methodologies

Due to the complexity of our system and its reliance on network communication, we have split our testing strategy into three parts:

1. Automated testing - Use automated scripts and runnable java classes to test individual modules for correctness.
2. Manual insert/remove testing - Use test cases and some manual setup to ensure that the overall network behaves correctly.
3. STEP graph testing - Use provided STEP programs to ensure the reliability of the system.

Analysis of the types of functions that our hashing system carries out quickly made it clear that robust testing of this system is both time consuming and operationally complicated. Simple scripts will not allow the type of verification that we require.

There are three basic modules at work.

1. The client SXE that only communicates to the hashing system via opcodes.
2. The client storage coupled with its proxy server.
3. The server storage and mediator.

Testing must not be isolated to a single module but also include the interaction among them. Testing the individual modules can be automated. We simply instantiate the different modules and run their APIs through tests which use a combination of black-box and glass-box strategies to find API vulnerabilities as well as other bugs exposed only through statement coverage. Although we cannot claim to create a set of inputs that ensure complete statement coverage, our tests come reasonably close with the added benefit of simplicity. We decided not to stub the server functions and to require the server to be running during these tests. The automated tests actually test both the modules as well as the communication between modules 2 and 3 above. Testing the interaction between modules 1 and 2 can be done using the opcodes via STEP graphs. This is why we decided to provide STEP graphs as both an example and as a way to test this API. Finally, our manual test cases are glass-box tests developed to test the mediator and storage as thoroughly as possible. It causes the server to go into recovery, update known subscribers, and also cause unknown clients to reinitialize and handshake with the server. Together these three methods of testing give us validation of individual modules, the interaction between those modules, as well as special cases. After validating these three items we believe that a user will find that further validation is unnecessary.

6 Reflections

Our project was constantly evolving. Although this was not intended, it is mostly contributed to our increasing familiarity and understanding of the snBench codebase as well as increased knowledge of accepted design patterns (which were highly applicable to the majority of the classes involved in our module!). Another major influence was our use of the Spiral Model as our process model throughout the semester. In this model we iteratively elicit requirements of the system, architect and design a suitable module, test the design through validation and verification and repeat. We quickly had working prototypes and were able to incrementally augment the system with new functionalities. As for the re-factoring of our code, we were able to apply numerous design patterns but after most development had already been completed. Specifically we made use of the Singleton Pattern whenever we needed to ensure the instantiation of only one instance of a class (eg., CentralizedHashStorage). We created a Mediator Pattern on the server that used both the Push and Pull patterns; furthermore, this was coupled with the Publish / Subscribe Pattern. We found that in so doing our code was easier to read and maintain, as well as exposing numerous bugs and limitations that were unanticipated (or caught by previous test cases). Our initial objective was to implement a fully distributed hashing scheme. However, after analyzing the access patterns typical in the problem domain we found that a central design would be sufficient and perhaps even preferable if there was sufficient caching.

Benefits, Costs and Limitations of our Design Choices

We chose to use a centralized design instead of a fully distributed design. The primary reasons for choosing the centralized model were the following: simplicity of design eases maintenance, equal or better query response, good design choice for networks of small memory node SXEs. Our code was re-factored with a lot of accepted Design Patterns that will be readily understandable improving the simplicity and ultimately the ease of maintenance in the future. The performance of our model equals or exceeds that of the typical distributed model. Since we employ caching, gets are typically very fast (requiring no network delays). Even in the worst case, only one network delay is incurred. The most scalable distributed hashing protocols has a logarithmic number (in the worst case) of network delays to find and route the hash entry. Our model is very applicable to sensor networks in which the participating nodes have small memory. A central resource with ample storage becomes requisite in such a situation. Some of the downsides to our model are that many demands are placed on the central resource. The central resource is a single point of failure. We feel this is reasonable since the Resource Monitor is also a single point of failure that could bring the network to a halt. A further requirement is for the central resource to have large physical memory, which is common for dedicated database servers. The performance of the central server could become a performance bottleneck since it doesn’t scale very well (linearly with number of nodes). A multi-threaded server listening socket would be a nice enhancement, but more considerations should be made for scalability (perhaps a dedicated server farm). The use of design patterns turned out to be a huge success. It made testing and maintenance much easier since these are abstractions which are commonly understood and “easy” to manage and test. We used a single threaded storage server. This choice was originally intended as a stepping stone to our final result but we ended up running out of time to implement a multi-threaded server which we could fully test. This has little benefit other than simplicity and avoiding possible race conditions if not careful. However, it does have the cost of severe lag in server response in some cases since on a worst case update the server will need to contact each client server before servicing any new requests. We chose a synchronous client-server design. The benefits of this are that each client is guaranteed
that when the hashing system returns, their input has been committed network-wide and that the data that they are receiving is globally up-to-date. The server becomes a bottleneck and there can be a serious performance cost to this, however. Again, in a worst case update scenario the client who did the update must wait for all subscribed clients to be updated before the put will fully succeed and return. This could cause lag in STEP code execution. If the server is unavailable long enough or dies, this synchronous relationship will cause system failure by way of the StorageConnectionException which is unchecked but thrown by the proxy under certain unrecoverable situations. We chose to leave the server as an in-memory database. There is therefore a single point-of-failure. We chose this because it was simpler to implement and could provide a stepping stone to adding persistence. It has the benefit of adding speed and simplicity to server lookups but has some serious limitations. The first limitation is that the system has a single point-of-failure. This is obviously not as robust as having a distributed framework but the Sensorium has other single points of failure, such as the Resource Manager, so a discussion with Michael convinced us that this did not limit the Sensorium in any new way. The second limitation of this is that the system has no guaranteed persistence. If the server should fail, all data is lost. Even if the server comes back up clients which reconnect are forced to reinitialize their cache to guarantee that they do not have stale data due to cached keys that they are no longer subscribed to.

7 Future work

There is plenty of work that could be done to both improve on our limitations as well as expand on how our current system works. Remove the synchronous relationship between the clients and server. This could be done in many ways. One could be by implementing a locking scheme which forces synchronous actions only when necessary. For example, anyone can change anything they want system wide with only the guarantee that their change will be committed but not in any transactionally consistent way. Ie, other clients may be updating this value as well so time-order is not guaranteed and any change could be overwritten before anyone sees it, even if the put succeeded. This basically translates to your cache not being guaranteed completely up to date for this value. If we add locking, a client could issue a lock() command which tells the server that you want exclusive use of that key. Once lock() returns you are guaranteed to have the most up-to-date value and control of it. After this point all other lock() calls for that key will hang until the key is unlocked. At which point the next lock() waiter will be granted their lock and given your change. Add persistence to the hashing. This could also be done in many ways. We suggest the use of adding an SQL backend to the server. All changes to the server could be treated as a write-through cache and committed to both the in-memory hashtable and the SQL database. If the server were to die and come back up it could get all system-wide key values and subscribers from this persistent storage and recovery would not be necessary. Add multi-threading to the server and remove that limitation. Perhaps create a pool of threads which service requests and another pool which publish subscription updates to clients. This could get complicated since they're using the same hashtable structures but if implemented correctly could greatly increase the performance of the system.

References