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

Dave Cecere, Ben Freiberg, Dustin Burke
Department of Computer Science
Boston University

## 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.

## Introduction

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, 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.

## Project Overview

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.”

## 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 codebase; 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 refactored 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.

## Schedule of Tasks

| | |
|---|---|
| Feb 27 | Completed Version 1.0, basic proof of concept |
| Mar 3 | Website launched |
| Mar 20 | Requirements and Architecture Documents completed |
| Mar 21 | Preliminary project presentation |
| Mar 24 | Completed Version 1.1, local hashing with snObjects as value |
| Mar 31 | Completed Version 2.0, central resource handles queries, local cache miss leads to global get |
| Apr 10 | Completion of math opcode library |
| Apr 14 | Project Plan readjusted to be in line with new objectives<br>Completed proof of concept method for cache validation |
| Apr 17 | Client/Server architecture complete, testing begins<br>Begin garbage collector<br>Need to handle client/server failures |
| Apr 19 | Code refactored with design patterns |
| Apr 21 | Completed Version 2.2, Cache validation has two protocols: cache invalidation or update messages from central resource, Implementation complete for "Hello" protocol designed to handle node failure and consistency of cache when either client or server fails |
| Apr 26 | Developed demo, email opcode complete – remains to be tested |
| Apr 27 | Tested email opcode, completed development of image processing opcodes, testing completed |
| May 1 | Tested demo |
| May 2 | Finished documentation, technical report |
| May 3 | Final presentation and Project Submission |
| | |

## Outline of Report

1. Components and Features of project
2. Organization of Java Packages
3. Quick User's Guide
4. Testing Methods – Validation and Verification
5. Discussion, Reflections and Conclusion

## 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.hash.put(String key, snObject value);
snObject sxe.core.hash.get(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, grayscale, thumbnail, load image, and save image. The motion detection 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 exceeds some threshold then report that motion has occurred. We discuss some limitations of this approach as well as proposed extensions in later sections.

# Organization of Java Packages

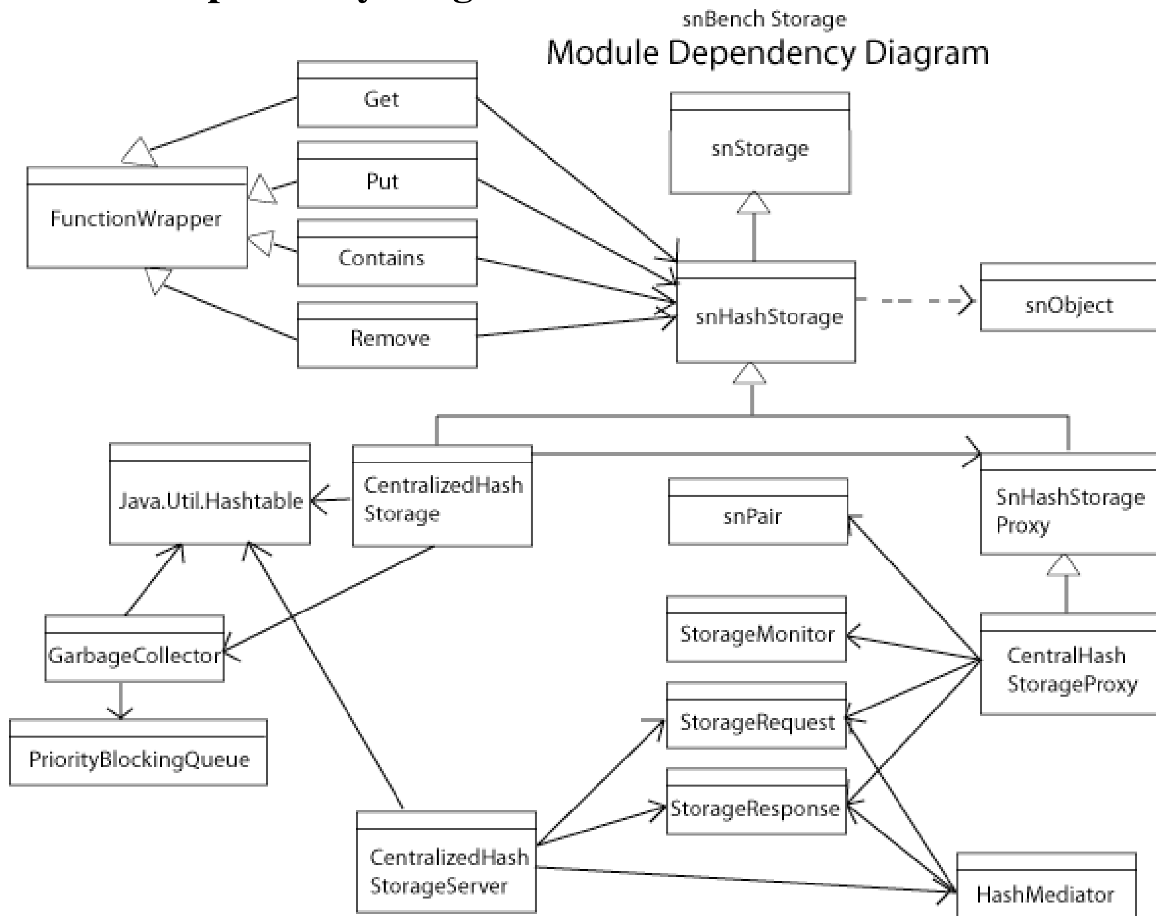
The organization of source files was very straightforward by grouping classes into packages. The following shows the hierarchy of directories and source files.

| | | |
|---|---|---|
| /snobject/ | snDouble.java | SnDouble type definition |
| /sxe/core/email | SendEmail.java | Outgoing email opcode |
| /sxe/core/hash/ | alert.java | Opcode to send message to console |
| | CentralHashServerProxy.java | Client-side proxy that handles communication with server |
| | CentralHashStorage.java | Local SXE storage which uses the CentralHashServerProxy to communicate with server. This is a singleton class |
| | CentralHashStorageServer.java | Runnable class on the server which implements the server listener and controls the server hashtable |
| | contains.java | Mapping of the Hashtable.containsKey() function to an opcode |
| | FlushCacheException.java | Exception thrown by proxy when server indicates that recovery has occurred and the |

| | | |
|---|---|---|
| | | client needs to reinitialize |
| | GarbageCollector.java | Reduces cache size in memory based on expiration |
| | GarbageCollectorItem.java | Items used by GarbageCollector |
| | get.java | Hashtable.get() mapped to opcode |
| | HashMediator.java | Implementation of the mediator pattern. This class assists in keeping track of subscribers. |
| | put.java | Hashtable.put() opcode mapping |
| | remove.java | Hashtable.remove() opcode |
| | SnHashStorage.java | The interface implemented to provide the client-side SXE's storage. |
| | SnHashStorageProxy.java | The interface implemented to created the SXE's local storage. |
| | SnStorage.java | The superclass of SnHashStorage. |
| | StorageConnectionException.java | Unchecked exception when a non-recoverable network error occurs |
| | StorageMonitor.java | Class which listens on each SXE for server updates |
| | StorageRequest.java | Object sent to begin communication between server and client |
| | StorageResponse.java | Response message sent to every request |
| /sxe/core/image/ | DetectMotion.java | Opcode to detect motion between two frames of snImages |
| | DiffTest.java | Test script of Image Diff |
| | GrayscaleTest.java | Test script of converting image to Grayscale |
| | ImageProcessing.java | Class containing suite of image processing functions |
| | loadSNImage.java | Opcode to load snImage from file |
| | MotionTest.java | Test script for DetectMotion |
| | saveSNImage.java | Opcode to save snImage to file |

| | | |
|---|---|---|
| | ThumbnailTest.java | Test script for creating thumbnail image |
| /sxe/core/math | | The filenames are self-evident and too numerous to list here |
| / | storagenetwork.xml | Configuration file for Storage |

## Module Dependency Diagram




## Data Model

Domains
Key:the set of keys (of type snString)
DataValue:the set of data values (of type snObject)
Client:set of IP addresses for the client machines

Relations
Maps:unique correspondence of Key item to DataValue item
Subscribes:subscribe client for the key item
Bind:client is bound to port (logically and physically connected)

Key +→! DataValue
Client *→* Key
Port ! -|→| * Client

## 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 opcodes we provide. A compelling example of the syntax can be found in the file "adt.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 sxe.Server http://localhost:8080 NONE

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

- 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 "storagenetwork.xml" along with the matching port number on which the server binds and listens. Next, open a third terminal window and type

- java sxe.Poster 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.

# Testing

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

## Automated testing

The script we created for this test is
/snbench/sxe/core/hash/testsuite/testWithServerRunning.sh
As the name suggests, you need to ensure that the server defined in snbench/storagenetwork.xml is up and running without any prior accesses. In the worst case there are three simple java programs to run with no parameters that you can execute one at a time.

You can launch the storage server with the following command (see CentralizedHashStorageServer documentation for more details):

java sxe.core.hash.CentralizedHashStorageServer <port> <updateType>

Once this is running simply execute the script from the command line and it will fire up the necessary tests. Output will be given indicating movement and success. The tests use the assert() call so that logic errors can be caught by checking which assertion failed.

---

## Running the test suite:

1.Start the CentralizedHashStorageServer on the machine and port indicated within the snbench/storagenetwork.xml file.
2.from your snbench directory run sxe/core/hash/testsuite/testWithServerRunning.
3.
4.Example output:

```
Start  Server:

(user) snbenchAFTER % java
sxe.core.hash.CentralizedHashStorageServer 1212 u
Initializing Storage Master
Storage Master running on local port 1212
Setting update message type to CACHEUPDATE
```

Run Tests:

Use the script provided or run the three individual test programs separately.

```
(user) snbenchAFTER % java -ea
sxe.core.hash.TestSnHashStorageProxy
Storage being initialized
Using centralized Storage on localhost port 1212 localport
= 3333
Registering with Server
Storage Initialized locally
Asserted that proxy was initialized successfully.
Asserted that get(), remove(), and containsKey() behave
correctly on empty storage
Asserted that putting a new key works successfully
Asserted that updating a key value is successful
Asserted that adding additional keys works correctly
Asserted that removing a key is successful
Asserted that removing the last key works successfully

(user) snbenchAFTER % java -ea
sxe.core.hash.TestSnHashStorage
Storage being initialized
Using centralized Storage on localhost port 1212 localport
= 3333
Registering with Server
Storage Initialized locally
Asserted that storage was initialized successfully.
Getting key=temp1
get missed locally with key=temp1
get missed globaly key=temp1
Removing key=temp1
Asserted that get(), remove(), and containsKey() behave
correctly on empty storage
Putting key=test1
Getting key=test1
found local cache of key=test1
Asserted that putting a new key works successfully
Putting key=test1
Getting key=test1
found local cache of key=test1
Asserted that updating a key value is successful
Putting key=test2
Getting key=test2
found local cache of key=test2
Getting key=test1
```

```
found local cache of key=test1
Asserted that adding additional keys works correctly
Removing key=test1
Getting key=test2
found local cache of key=test2
Getting key=test1
get missed locally with key=test1
get missed globaly key=test1
Asserted that removing a key is successful
Removing key=test2
Getting key=test2
get missed locally with key=test2
get missed globaly key=test2
Getting key=test1
get missed locally with key=test1
get missed globaly key=test1
Asserted that removing the last key works successfully

(user) snbenchAFTER % java -ea
sxe.core.hash.TestHashMediator
Asserted that construction was successfull
Subscribing testkey1 to key=127.0.0.1
Subscribing testkey1 to key=127.0.0.1
Client testkey1 found in subscribers table for
key=127.0.0.1,not added
Subscribing testkey1 to key=127.0.0.1
Client testkey1 found in subscribers table for
key=127.0.0.1,not added
Subscribing testkey1 to key=127.0.0.1
Client testkey1 found in subscribers table for
key=127.0.0.1,not added
Completed all tests for testHashMediator()
```

## Manual testing

This section is a bit more tedious but necessary to really test the functionality of the overall system. There are a number of manual test cases to follow. For ease of use, we created an html front-end that allows you to insert and delete keys (integer and string only) from any given client. You can reach this webpage after starting an SXE by going to "<SXE server address>/snbench/sxe/storage/". The trailing slash is required. The web interface is intended to simply test that values are being seen by the appropriate systems so it is limited to only strings and integers. Any string that does not parse to an integer will be treated as a string, applying to floats and doubles.

SETUP

1. Start a CentralizedHashStorageServer
2. Configure two different SXE's to point to that server by changing their snbench/storagenetwork.xml file.

TESTCASES (do all of below using both 'u' and 'i' startup parameters on server):

1. Simple put/get from two different clients.
    a. On SXE A, do a put where key="test1" and value="A"
    b. On SXE B, do a get where key="test1"
    c. Confirm that B now sees value "A" for key "test1"
2. Update/Invalidation
    a. Starting where we left off from test #1 you should have (test1,A) on both SXE's. From B, do put("test1","B").
    b. You should see activity on the server and depending on your update pattern A should either flush its value or update its value. You can confirm this by going back to the "/storage/" page on A and looking at the values or simply doing a 'get' for "test1"
3. Repeat Test #2 with a removal and ensure that both SXE's no longer have the (key,data) pair.
4. Recovery
    a. restart the CentralizedHashStorageServer
    b. restart an SXE
    c. Create a bunch of keys from that SXE
    d. Kill the StorageServer and start it backup
    e. Go any action from the SXE which requires a trip to the server (put will always work).
    f. You should see output on the server showing a handshake as the SXE is reinitialized. Refresh the /storage/ page and you will see that all keys that it had were flushed except for the value it just put/get in step E.
5. Server Failure
    a. Start a Server and SXE
    b. Bring a few keys to the SXE through gets or puts
    c. Disable the server
    d. All future operations that require the server will timeout and the system will generate StorageConnectionExceptions. This is an unchecked exception that is currently a system failure.

STEP graph testing

Step graphs to use (found in snbench/testsuite folder):

1. testput.xml
2. testget.xml
3. testremove.xml
4. testcontains.xml

Vary the ordering when POST-ing these STEP files (via the sxe.Poster) to see their different results. We made this simple on purpose so that you will need to watch the Server console to see the different values the opcode nodes evaluate to.

## What was our testing approach and why did we choose these tests?

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.

## Reflections on the semester's progression

Our project was constantly evolving. Although this was not intended, it is mostly contributed to our increasing familiarity and understanding of the snBench code-base 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 (e.g., CentralizedHashStorage). We created a Moderator 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 exposed 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 refactored 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 because 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.

## 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 transactionaly 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


[1] A. Kfoury A. Bestavros, A. Bradley and M. Ocean. snBench: A development and run-time platform for rapid deployment of sensor network applications.

[2] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification and Object-Oriented Design.* Addison Wesley, 2001.