The Scalable Hyperlink Store

Release Note
Version 1.0
December 18, 2013

Marc Najork

Overview

The document describes release 1.0 of the Scalable Hyperlink Store (SHS), available in source and binary form from research.microsoft.com under the Microsoft Research Academic License. SHS is a specialized storage service that provides access to web graphs – graphs induced by web pages and the hyperlinks between them. SHS compresses the graphs by leveraging empirically observed properties of web graphs (namely, the prevalence of relative links, i.e. links between pages on the same web server), it keeps the graph in main memory to provide fast access, and it partitions the graph over multiple machines to provide scalability. The original SHS design is described in [1]. Subsequently, I evolved SHS to allow clients to maintain transient state in the service (say, the score of each web page during the course of a PageRank computation); this addition is briefly described in [2]. Since then, SHS has evolved further, particularly in how it provides fault tolerance. The remainder of this document explains how to set up an SHS service, gives a brief overview of the API, sketches the new fault tolerance scheme, and concludes by outlining my plans for the next release.

A word about the SHS code

The current version of SHS is written in C#. The release includes a Visual Studio 2013 "solutions" file, and I would strongly encourage you to use Visual Studio to write SHS client applications. Visual Studio Express is available at no cost to the general public; Visual Studio Professional is available at no cost to students under the Microsoft DreamSpark program. The code is structured into several "projects", with each project in a separate folder. The "Library" project contains the library that SHS client applications link to, the "Server" project contains the SHS server, and the other projects contain SHS client applications that illustrate various capabilities or that are useful for building and maintaining SHS stores. In particular, the release contains the SHS implementations of the five algorithms benchmarked in [2] – PageRank, SALSA, Approximate Shortest Path, Weakly Connected Components, and Strongly Connected Components.

I use Code Contracts [3] to express assertions, preconditions, postconditions and object invariants. In order to compile code using Code Contracts, you need to download "Code Contracts for .NET" from the MSDN web site and install it; you might also want to download and install the "Code Contracts Editor Extensions". You don't need to install Code Contracts unless you want to recompile the SHS server or library.

Setting up an SHS service

SHS maintains a service to store and access web graphs as well as transient per-vertex data used by a client's computation. SHS runs on multiple servers, one of which is called the leader and is responsible for orchestrating interactions between clients and servers. To start an SHS service, download and

unpack the code release, compile the code or use the provided pre-compiled binaries, and start the service by running the following command on each computer supposed to provide the service:

SHS.Server *leader*

where leader is the name of the computer that is supposed to be the leader. Once started, each server attempts to contact the leader to advertise its availability. The leader keeps track of which servers are currently available, and it maintains the overall state of the system – what stores exist and so on.

In order to test whether the service is running, you can run one of the regression tests that are included with the release:

SHS.RegressionTest2 *leader* 10000 100

This test creates a synthetic web graph with 10000 URLs (as per the first command-line argument) and initially no links between them on the client side, and then repeatedly (100 times in this case, as per the second command-line argument) mutates the graph by adding new and deleting existing links, saving the graph in the SHS service, and testing whether the graph in the service and the local graph agree.

By default, this regression test creates an SHS store that tolerates one server failure, so you should provide at least two machines. If a non-leader server dies, the system reconfigures and service continues. If the leader dies, service is interrupted until the leader is restarted, but the client should not crash.

Assuming that a non-leader server has failed and cannot be restarted, you can bring in a new server to take its place by running the following command on the new server:

SHS.Server *leader failedserver*

The new server contacts the leader, advertises its availability, and offers to take the place of *failedserver*. The leader checks if *failedserver* is truly unavailable, update its meta-data if that is the case, and provisions the new server with all data that was on *failedserver*.

Another handy command is SHS.ManageStores, which allows you to list all stores maintained by an SHS service and to delete selected stores. Run SHS.ManageStores without any arguments to get usage information (in general, all SHS programs print usage information if invoked without arguments).

Finally, there are two programs that are useful for populating a store: SHS.ConvertCwGraphToShsInput reads a graph encoded in the ClueWeb graph file format [4,5] and writes it out in a format that more closely resembles the output of a web crawl, and SHS.Builder reads a file in that latter format and creates a new SHS store containing that graph. Together, these two programs allow you to create SHS stores containing any of the ClueWeb graphs.

Assumptions about faults and resulting fault tolerance guarantees

I made a number of assumptions about which types of faults I should deal with, and (try to) provide various guarantees on how well SHS tolerates such faults. The assumptions are:

- All server failures are crash failures. I do not deal with “soft” failures caused by bit-flips in memory or corruption of data coming from disk or network. However, I aim to cope with truncated files, caused for example by a server dying while writing data to disk.

- Networks are reliable. While servers may die, I assume that the network does not get partitioned, i.e. that running server processes can communicate with each other. This is a somewhat unrealistic assumption; for example, a top-of-rack switch may fail. Actually, I did consider the possibility in much of the implementation, but I have not tested for it. (See below for a description of the testing methodology I used.)
- Non-leader servers may fail without prolonged service interruption. Associated with each store is a replication factor *r* (at least 1, and 2 by default). SHS tolerates up to *r*-1 non-leader servers failing. If such a failure occurs, a client's store clerk will ask the leader for a set of surviving servers capable of providing the store. The failure is transparent to the client application unless the service is maintaining job-specific state for the client.
- Failure of the leader causes service interruption, but clients do not crash. Service will resume once the leader is brought back on line. In other words, the leader is a single-point-of-failure, its meta-data is not replicated, and it is up to you to guarantee durability of that meta-data (for example by storing it on a RAID volume). One of my future-work items is to replace this non-replicated meta-data by a replicated state machine and thus eliminating the single-point-of-failure.

The API for SHS client applications

SHS provides a few basic abstractions to clients. Here is an overview; you may also want to refer to [1].

Clients access the SHS service by creating a Service object, which has the following public members:

- Service(string *leader*) – constructs a new Service object for an SHS service orchestrated by the server named *leader*.
- NumServers() – returns the number of SHS servers currently available.
- ListStores() – returns an array of StoreInfo values, one for each store maintained by this service. Think of it as the equivalent of a directory listing, and see SHS.ManageStores for an example use. The most important member of StoreInfo is ID, which uniquely names each store.
- CreateStore(int *p*, int *r*) – creates and returns a Store object (see below) for a new SHS store to contain a web graph, with a replication factor of *r* (i.e. tolerating up to *r*-1 failed servers) and sliced into *p* partitions, thus replicating its persisted state across *p*+*r*-1 servers. This method has three additional optional parameters: *friendlyName*, a way for users to assign memorable names (descriptions) to stores; and *numPartitionBits* and *numRelativeBits* that control a trade-off between compression on one hand and the maximum number of partitions and URLs per partition on the other. The default value of *numPartitionBits* is 4, allowing for at most 2^4 partitions; and the default value of *numRelativeBits* is 32, allowing for 2^{32} URLs per partition.
- CreateStore() – creates and returns a Store object for a new store with a replication factor of 2 (i.e. tolerant to one server failure) and sliced into *n*-1 partitions (where *n* is the number of available machines), thus replicating its persisted state across all *n* machines.
- OpenStore(Guid *storeID*) – returns a Store object for an existing store, identified by its unique identifier *storeID*.
- DeleteStore(Guid storeID) – deletes an existing store identified by *storeID*.

A Store object represents a web graph stored by the SHS service. It has the following public members:

- ID – the globally unique identifier of this store, assigned by CreateStore, discoverable by ListStores, and used by OpenStore and DeleteStore (all above).
- AddPageLinks(IEnumerator<PageLinks> *b*) – add a batch of PageLinks values (each one containing the URL of a web page and of all the links in that page) to the store. Doing so obviously changes the store, which might impact concurrent uses of the same store by other threads or other clients; more below. AddPageLinks returns a number identifying the "epoch" of the updated store.
- Seal() – "seals" the store, making it impossible to perform any future AddPageLinks operations on it.
- Close() – closes the store, releasing any server-side resources associated with it.
- Request(SubStore *s*) – provides a hint to the service that this client will use one of this store's "substores" (URLs, forwards links, backward links) soon, so the service can prefetch it.
- Relinquish(SubStore *s*) -- provides a hint to the service that this client is done with a substore for the time being, allowing the service to page it out of main memory if there are no other users.
- MarkAtomic() – indicates the beginning of a sequence of method calls that are meant to be atomic – that is, the graph (and its epoch) should not change while they are in progress. If the graph does change, the next method call will throw an EpochChanged exception, and it is up to the client application to catch and handle it – for example by restarting the "transaction", beginning at MarkAtomic. All Store methods described from here on may throw EpochChanged. Each of these methods has a variant that takes an explicit epoch number as a ref parameter. SHS.RegressionTest3 provides an example on how to use them, and a subtle example of a situation where MarkAtomic won't do and explicit epoch numbers must be used.
- NumUrls() – returns the number of URLs in the store.
- NumLinks() – returns the number of links in the store.
- MaxDegree(Dir *d*) – return the maximum out-degree (*d*=Dir.Fwd) or in-degree (*d*=Dir.Bwd) over all vertices of the graph represented by this store.
- UrlToUid(string *url*) – takes a URL contained in this store and returns a UID ("unique identifier"), a number representing *url* ***during the current epoch***. Returns -1 if the store does not contain *url*.
- UidToUrl(long *uid*) – takes a UID and returns the URL it represents.
- BatchedUrlToUid(string[] *urls*) – a variant of UrlToUid taking a batch of URLs.
- BatchedUidToUrl(long[] *uids*) – a variant of UidToUrl taking a batch of UIDs.
- Uids() – returns an IEnumerable of the UIDs representing every URL contained in this store. Useful for "foreach" loops.
- IsLastUid(long *uid*) – true if and only if *uid* is the last UID in the IEnumerable returned by Uids(). Useful as a termination criterion when grouping the result of Uids() into batches, e.g. for BatchedUidToUrl. Accompanied by the helper class Batch, see SHS.PageRank for an example on how to use both.
- GetDegree(long *uid*, Dir *d*) – returns the out-degree (*d*=Dir.Fwd) or in-degree (*d*=Dir.Bwd) of the vertex represented by *uid*.
- BatchedGetDegree(long[] *uids*, Dir *d*) – a variant of GetDegree that takes a batch of UIDs.
- GetLinks(long *uid*, Dir *d*) – returns the incoming (*d*=Dir.Bwd) or outgoing (*d*=Dir.Fwd) links of the vertex represented by *uid*.

- BatchedGetLinks(long[] *uids*, Dir *d*) – a variant of GetLinks that takes a batch of UIDs.
- SampleLinks(long *uid*, Dir *d*, int *n*, bool *consistent*) – sample *n* of the incoming or outgoing links of *uid* (or return all if there are fewer than *n*) using either random or consistent sampling.
- AllocateUidState<T>() – ask the service to associate a variable of type T with each vertex in the graph, and returns a UidState object to access these server-side variables. Requires the store to be sealed. T must be one of the following 14 primitive types: bool, byte, sbyte, ushort, short, uint, int, ulong, long, float, double, decimal, char, or string (there is rudimentary support for general types, but it is not there yet ...). See below for details on how server-side variables are read and written. If a value is written to a server-side variable, the store is marked as "dirty".
- Checkpoint() – checkpoints all server-side variables associated with this Store, and removes any "dirty" marks from the store. If the system undergoes any reconfiguration (because a server crashed) while the store is "dirty", all server-side variables associated with the store are reset to their values at the last checkpoint, and a ServerFailure exception is thrown. It is up to the client application to catch and handle the exception – for example by restarting the computation at the state of the last checkpoint. See SHS.PageRankFT for an example.

Store provides three more methods (SetUrlToUidCacheParams, PrintCacheStats, UidToLid) which I won't describe here, and which are on the verge of being declared obsolete. Speaking of which, the following five methods described in [1] have already been removed from the API:

- MinUid(int *partitionID*)
- MaxUid(int *partitionID*)
- NumUrls(int *partitionID*)
- NumLinks(int *partitionID*)
- MaxDegree(Dir *d*, int *partitionID*)

While easy to implement, all of these methods broke the abstraction boundary between client and service – clients should not care which partition holds which UID.

A UidState<T> encapsulates client-specific server-side state associated with each UID (vertex). UidState objects are created by calling Store.AllocateUidState (see above). The class has six public methods:

- Get(long uid) – get the value of the server-side variable associated with vertex *uid*.
- Set(long uid, T val) – set the variable associated with *uid* to *val*.
- GetMany(long[] uids) – a variant of Get that takes a batch of UIDs.
- SetMany(long[] uids, T[] vals) – a variant of Set that sets the values of a batch of UIDs.
- GetAll() – returns an IEnumerable of UID/value pairs for all UIDs in the store.
- SetAll(Func<long,T> f) – updates the variable associated each UID in the store to a value computed by applying function f to the UID.

The UidState abstraction is tricky to program against. GetMany and SetMany calls tend to be paired (read the values of variables associated with a set of vertices and write back updated values), but if there are multiple occurrences of the same UID in the same batch, the value of the last occurrence will override all previous updates. I provide a helper class UidMap to make it easier to tackle such situations (see SHS.PageRank for an example on how to use it), but I plan to provide a much simpler abstraction in the next release – so please don't get wedded to the current API!

Changes from the SHS version described at Hypertext 2009

The current version of SHS differs from the one described in [1] in several ways, some of which are visible to the client and some of which are not. Visible changes include the refactoring of "ShsClerk" into "SHS.Service" and "SHS.Store", the elimination of five Store methods (that became obsolete with the introduction of the Uids() enumerator), support for server-side state (leveraged in [2]), and a checkpointing facility for server-side state that was added very recently.

The most significant change "behind the scenes" is how a store is partitioned across machines. The original design described in [1] assumed that, given *n* servers and a replication factor *r* (guarding against up to *r*-1 failed servers), a store is partitioned into *p*=*n* partitions, and each partition is replicated onto *r* servers. Since the service needs *n* servers to serve up these *p*=*n* partitions, the store is partitioned a second time into *n*-*r*+1 partitions, each partition again replicated onto *r* servers, meaning that even if r-1 servers fail, there will be enough surviving servers to serve the (*n*-*r*+1)-partition store, and there will be a replica of each partition on some surviving server. One immediate consequence of this design is that the two stores (called "normal-mode" and "degraded-mode" store in [1]) have a different mapping between URLs and UIDs. While it is possible (and actually reasonably efficient) to map from one UID space to another, it greatly complicates the code.

The introduction of server-side client-specific state (described in [2]) and the machinery for checkpointing and replicating that state to make it fault-tolerant as well caused me to rethink the basic partitioning scheme. In the current release, given *n* servers and a replication factor *r* (guarding against up to *r*-1 failed servers), a store is partitioned into *p*=*n*-*r*+*1* partitions, and each partition is replicated onto *r* servers. This means that even if *r*-1 servers fail, there are still *p* servers left, enough to serve up the *p* partitions. This obviates the need for maintaining a separate "degraded-mode" store, and the need for translating UIDs between "normal" and "degraded" UID spaces, and it incidentally means that UIDs have a single partition ID, as opposed to a "normal-mode" and "degraded-mode" partition ID as described in [1]. This also removes a great deal of complication from the code.

There is one feature described in [1] that is absent from the current release: the ability to change the number of servers, as described in the last part of section 4 of [1]. At this point, I am not planning to add this functionality – it greatly complicates the system and impacts performance.

A note about testing

In addition to running the programs I benchmarked for [2], I tested the code using the following regression tests:

- SHS.RegressionTest1 – reads a file in the format produced by SHS.ConvertCwGraphToShsInput, creates an SHS store from it, and exhaustively verifies that the graph in the store matches the graph in the file. Good for testing the major methods on large, real graphs, with URLs that may contain Unicode characters etc.
- SHS.RegressionTest2 – already described above. Maintains a smallish graph in memory, and repeatedly mutates it, propagates that change to the SHS service, and verifies that the local graph and the graph maintained by SHS remain in sync. Good for testing link addition and deletion (the latter not being tested by RegressionTest1)

- SHS.RegressionTest3 – similar to RegressionTest2, but spawns multiple threads to each compare the two graphs. Good for testing thread safely (both on the client and the servers) and for exercising the EpochPassed machinery.

To test service reconfiguration and fail-over, I wrapped the SHS.Server invocation into a DOS loop that restarts the server whenever it terminates:

```
FOR /L %x IN (0,0,1) DO SHS.Server leader
```

I then ran a Powershell script ("kill-servers.ps1") that round-robins over servers and kills one of them every five seconds. The script is:

```
$servers = "svc-d1-02","svc-d1-03","svc-d1-04","svc-d1-05"
While ($TRUE) {
  ForEach ($x in $servers) {
    Invoke-Command -ComputerName $x -ScriptBlock { Stop-Process -Name SHS.Server }
    Start-Sleep -Seconds 5
  }
}
```

In order for this to work, you first need to enable PowerShell remoting by running the following PowerShell command on each server:

```
Enable-PSRemoting -Force
```

Killing an SHS.Server falls short of power-cycling the underlying server, so there may be bugs that are not discovered by this methodology. Still, it goes a long way, and it enabled me to run duty cycles with thousands of kills & restarts.

Future work

My future-work list consists of four groups of items.

- "Code cosmetics": I have a long laundry list of things that can be made prettier, simpler, or more efficient. None of them should impact client applications.
- "Rethinking UidState": The UidState API is difficult to program against, and the resulting code is hard to read. I have some ideas on how to make the API much simpler and more natural, at hopefully only moderate cost in performance. If this works out, I will replace GetMany, SetMany, GetAll and SetAll with this new machinery. Any client using these calls would need to be changed.
- "Rethinking URLs and UIDs": In the more distant future, I might rethink whether URLs and UIDs should be distinct concepts to client applications, or whether I can hide the distinction at least to some extent.
- Removing the single-point-of-failure stemming from the fact that the system's meta-data is not replicated but kept by a single leader.

And of course, I'm planning to fix bugs as I discover them. If you find one, please let me know at najork@microsoft.com or marc@najork.org. For that matter, please let me know if you find SHS useful!

References

[1] Marc Najork. The Scalable Hyperlink Store. 20th ACM Conference on Hypertext and Hypermedia, 2009.

[2] Marc Najork, Dennis Fetterly, Alan Halverson, Krishnaram Kenthapadi and Sreenivas Gollapudi. Of Hammers and Nails: An Empirical Comparison of Three Paradigms for Processing Large Graphs. 5th ACM Intl. Conference on Web Search and Data Mining, 2012.

[3] Code Contracts home page. http://research.microsoft.com/en-us/projects/contracts/

[4] ClueWeb09 home page. http://lemurproject.org/clueweb09/

[5] ClueWeb12 home page. http://lemurproject.org/clueweb12/

Appendix

What follows is a transcript of creating an SHS store on four primary servers plus one secondary server containing the ClueWeb09 Category B web graph [4] and running 100 iterations of the weakly-connected component algorithm described in [2] over it:

```
d:\najork\shs-testing>SHS.ConvertCwGraphToShsInput ClueWeb09_WG_50m_NodeList_Full.txt.gz
ClueWeb09_WG_50m.graph-txt.gz cw09-catb.bin.gz
Expecting 428136613 body lines
Read graph file, found 428136613 nodes and 454075604 edges.
Wrote out result file; starting sanity check.
Found 428136613 pages and 454075604 links. Job took 17235.476 seconds.

d:\najork\shs-testing>SHS.Builder svc-d1-02 cw09-catb.bin.gz "ClueWeb09 Category B"
SHS service is currently running on 5 servers
### [connect 1] Opened partitions on SVC-D1-02 SVC-D1-04 SVC-D1-03 SVC-D1-05
Done. Building store 12c299f4e1a34e21a50412cd0d8a1f92 took 36414.403 seconds.

d:\najork\shs-testing>SHS.WCC svc-d1-02 12c299f4e1a34e21a50412cd0d8a1f92
### [connect 1] Opened partitions on SVC-D1-02 SVC-D1-04 SVC-D1-03 SVC-D1-05
Done. Job took 2855.371 seconds.
```