Shared – 0.1.0

Kevin Jones (https://twitter.com/hutkev)

Shared implements a Node/Javascript data access API for data stores that support atomic operations. A simple in-memory data store is included for use within a node “cluster” (a set of node processes running within a single machine that communicate via pipes).

The Shared API borrows from work on transactional memory by providing an 'atomic' operation which is used to encapsulate updates to a data store which are written in idiomatic Javascript:

```
function savingsToCurrent(amount) {
    store.atomic(function(db) {
        if (db.savings > amount) {
            db.saving -= amount;
            db.current += amount;
            if (db.history === undefined)
                db.history = [];
            db.history.push({transfer: amount, from: 'saving',  to: 'current' });
        }
    });
}
```

The closure passed to the atomic function is given access to a view of the data store via the 'db' object. The view is a partial cache of the primary store held on the master node process. After the closure has completed running a readset and writeset are transmitted to the primary data store to be committed as a single operation. If the operation can not be committed because the local cache was out of date with respect to the primary store then the local cache is updated and the closure is re-executed.

The in-built memory store runs on a single thread and so naturally provides consistency & isolation between operations. A durable store will be available in a future version. The store is structured as a graph of the JavaScript composite Object and Array datatypes with a well known root object, the 'db' value in the example. There are no schema restrictions so objects and arrays may be added during any atomic operation, these changes also being made atomically. See Restrictions later for more information and data type handling.

Care should be taken when placing code which has side-effects inside an atomic call. This may be executed multiple times resulting in undesired behaviour.

Creating a Store

The atomic function is available on Store objects that should be created in each process. Each Store object maintains a partial cache so best practice is to create only one for each process to minimize memory use although multiples Stores within a single process can be used if needed. A primary store is automatically created in the node cluster master process when the first Store is created. This should be started before forking children to ensure the primary store is initialized before it can be used. A simple template for cluster operation is:

```
var cluster = require('cluster');
var store = require('shared.js').createStore();
```

```
if (cluster.isMaster) {
   store.atomic(function (db) {
      // Init store structure
   });
   cluster.fork();
} else {
   store.atomic(function (db) {
      // Worker action
   }, function (err) {
      // err is null on success
      process.exit(0);
   });
}
```

Atomic Operation

Due to the use of message passing the atomic operation is asynchronous and so supports a callback model that should always be used to handle additional processing. The signature in TypeScript is:

```
atomic(
   handler: (store: any) => any,
   callback?: (error: any, arg: any) => void
): void;
```

The handler closure is passed a reference to the store. Any return value from the handler closure is passed as the second argument to the optional callback closure, this value has no bearing on the atomic operation itself. The callback also receives an error object, this will be null if the atomic operation completed successfully, otherwise it will contain diagnostic information.

Exceptions thrown in the handler are caught automatically and passed as the error object of the callback closure if one has been provided. Any changes made prior to an exception been thrown will not be committed and so exceptions provide a convenient way to terminate an atomic operation abnormally.

Restrictions

The objects held within the store are managed by the store using techniques similar to the JavaScript proposal for Object.observe. These techniques and the library design impose some restrictions that must be observed for correct operation:

- Do not leak objects from the handler closure to non-atomic code, the objects must only be accessed from within an atomic block.
- Do not rely on an objects prototype. Objects may be constructed with 'new' inside an atomic block but the prototype of an object is not recorded so it will become a plain old object after the changes are committed.
- Do not define getter/setter methods on objects, these are used in parts of the implementation and will cause the atomic code to break.
- Avoid assigning functions to object properties, these are currently silently ignored.
- Date types may be used but they are currently mapped to Strings during a commit.

- In this version avoid excessive growth of the data graph. The library requires a garbage collector to recover dead objects from its own data structures and this is not fully implemented.

Performance Considerations

The library is using a form of optimistic concurrency control. This means it runs without locking which is helpful in the presence of failing nodes but that areas of high contention in the data store can significantly slow system progress. When considering how to design data held within the store it is best to try an isolate writers from each other as much as possible. For example, if you were logging data into an array then it may make sense to create an array for each writer/worker and a summary process that periodically combines these array into a consolidated log when system load is low.

In general it is best to split the schema into the smallest possible units to achieve best performance as the library treats each Object and Array as a separately version-able item. Array operations are also defined to optimise common changes, such as push, pop, shift, unshift, splice and reverse. These will operate quickly but operations involving sorts or array retrieval may perform poorly, specifically when there are conflicts that require new versions of large Objects and Arrays to be serialized.

The base performance of the library is largely determined by the speed of the messaging passing layer but currently there are also a number of other areas where the library could be improved to both minimize the cost of processing messages and the number of messages being used when conflicts occur. I would expect the performance of both non-conflict and conflict scenarios to improve significantly in future versions.

Examples

The distribution includes some examples you may want to review:

- Counters – A simple count down example where multiple workers each decrement their own counter in the store until it reaches 0.
- Bank – A bank account money transfer simulation. A set of workers performs a set number of transfers between randomly selected accounts. Success is not losing any of the money!
- Work – Workers pull requests to computer Fibonacci numbers from a simple shared queue and post their results on a second queue.

License & Source

There is no license currently applicable to this code, you require the Authors permission to use it in any form.

The source code (~100kB in 3.3kLOC) is written in TypeScript and compiled to JavaScript and then minified with the Google Closure compiler. This distribution format is somewhat experimental because it's not clear how robust this toolchain is although it does produce small code (~41kB), the TypeScript compiler is very new. The release form does however pass all the unit tests and no issues were discovered in the TypeScript compiler during development.