AtomizeJS
Safe Distributed Shared Objects

Matthew Sackman
matthew@rabbitmq.com

Introduction

Trends of developers

- Push more and more logic client-side
- *Single Page Website / Application*
- Substantially reduce application logic on server
- Server reduced to security, marshalling data, and distributing data

Trends of developers

- Push more and more logic client-side
- *Single Page Website / Application*
- Substantially reduce application logic on server
- Server reduced to security, marshalling data, and distributing data
- Various techniques for safely modifying and distributing shared data

Trends of developers

- Push more and more logic client-side
- *Single Page Website / Application*
- Substantially reduce application logic on server
- Server reduced to security, marshalling data, and distributing data
- Various techniques for safely modifying and distributing shared data:
  Remote Procedure Call, Polling and distributing, others

Familiar Problems?

- NodeJS being single-threaded is broadly welcomed: simplifies development
- But lots of clients making changes to shared data-structures: back to the world of concurrency and parallelism
- What sort of problems might we want to solve?

Familiar Problems?

- NodeJS being single-threaded is broadly welcomed: simplifies development
- But lots of clients making changes to shared data-structures: back to the world of concurrency and parallelism
- What sort of problems might we want to solve?
  Example: inserting our username into an object

Familiar Problems?

- NodeJS being single-threaded is broadly welcomed: simplifies development
- But lots of clients making changes to shared data-structures: back to the world of concurrency and parallelism
- What sort of problems might we want to solve?
  Example: inserting our username into an object:

```
var users = {}; // Somehow populated by server
function register (myName) {
  if (myName in users) {
    alert("Username " + myName +
          " already taken, try again");
  } else {
    users[myName] = {};
    // Assume this change then gets sent to server
  }}
```

Familiar Problems?

- Races! The inspection of `users` and the modification allow for the `users` object to change in between.
- Sure, maybe not in a single browser (due to JS being single-threaded), but several browsers running the same code at the same time?
- How do you even detect this sort of collision?
- If all you can do is detect a collision after the event, can you still solve this sort of problem?

NowJS

- Has a timer which every 1000ms (or longer!) processes the *distributed object* `now`
- Essentially calculates the diff between the previous version and the current version of the `now` object
- Also uses functional getters and setters to intercept changes to variables: not a true proxy so a bit limited
- But explicitly avoids conflict resolution:

> *Note that you can write properties and call functions with the `everyone.now` object but you cannot read values. Since `everyone.now` represents multiple clients, they can have different values so reading them doesn't make sense.*

- Instead, RPC is used to invoke such functions on the server only

Meteor

- Meteor's `livedata` package heavily coupled to `mongo`. Collections are the shared storage
- Client-side implementation of (subset of) `mongo` called `minimongo`
- Modifications to client-side collections also send RPC calls to server
- The server can send down to clients modifications to collections
- Still multiple clients can attempt to modify the same collection: both do an insert of the same key but different values. Which wins? How can you tell?

What's really needed to solve these problems?

- Locks

What's really needed to solve these problems?

- Locks are well studied

What's really needed to solve these problems?

- Locks are fairly well hated too

What's really needed to solve these problems?

- Locks are fairly well hated too
- But locks are also used in the implementation of *transactions*, and we all know how to use transactions

What's really needed to solve these problems?

- Locks are fairly well hated too
- But locks are also used in the implementation of *transactions*, and we all know how to use transactions
- Enter, Software Transactional Memory

Software Transactional Memory (STM)

- Just like database transactions, you write transactions in your code
- These transactions are applied to the shared state, somehow, maintaining (some of) the *ACID* properties
- atomically: Transactions are atomic (all or nothing)
- in isolation: Transactions are isolated from one another. That is, even though in general there will be many transactions running concurrently, any given transaction's updates are concealed from all the rest, until that transaction commits. Another way of saying that same thing is that, for any two distinct transactions T1 and T2, T1 might see T2's updates (after T2 has committed) or T2 might see T1's updates (after T1 has committed), but certainly not both.

Example

```
function register (myName) {
  atomize.atomically(function () { // The Transaction
    if (myName in atomize.root.users) {
      return false;
    } else {
      atomize.root.users[myName] = atomize.lift({});
      return true;
    }
  }, function (success) {          // The Continuation
    if (!success) {
      alert("Username " + myName +
            " already taken, try again");
    }
  });
}
```

Advantages

- No explicit locking!
- Cannot deadlock!
- Lots of optimisation opportunities
- Performance can match the most perfect fine-grained locking equivalents

Implementation - one approach of many

- Client creates empty *transaction log*
- Client runs transaction and captures in the *transaction log* the effect of the transaction along with the version of every object read or written to
- Client sends transaction log to server
- If object versions are *current* according to the server, apply transaction on server and return `success` to client
- Otherwise:
  - Don't modify anything on the server
  - Send updated objects to client along with `failure` message
  - Get client to throw away old transaction log (i.e. undo the effect of the transaction) and rerun the transaction (i.e. `goto 10`)

STM in JavaScript

- So transactions are run client-side, but then the effect is sent to the server and verified. Thus transactions run in clients in parallel.
- Any transaction that reads an *old* version of an object will get restarted with an updated copy of that object
- The continuation only gets invoked once the transaction function has been run and committed - i.e. it completed without anyone else modifying any of the objects that it read or wrote to

Transactions on steroids

- `retry`: this allows you to say *abandon this transaction, but restart it when someone modifies any of the variables I've read so far*
- `orElse`: this allows you to compose transactions easily: provide a list of transaction functions and when one hits a `retry`, just start the next transaction function instead of doing a full `retry`
- `retry` allows you to implement the *observer* pattern, and from there, you can build out e.g. shared queues
- Unlike databases, transactions are automatically restarted

In comparison to locks

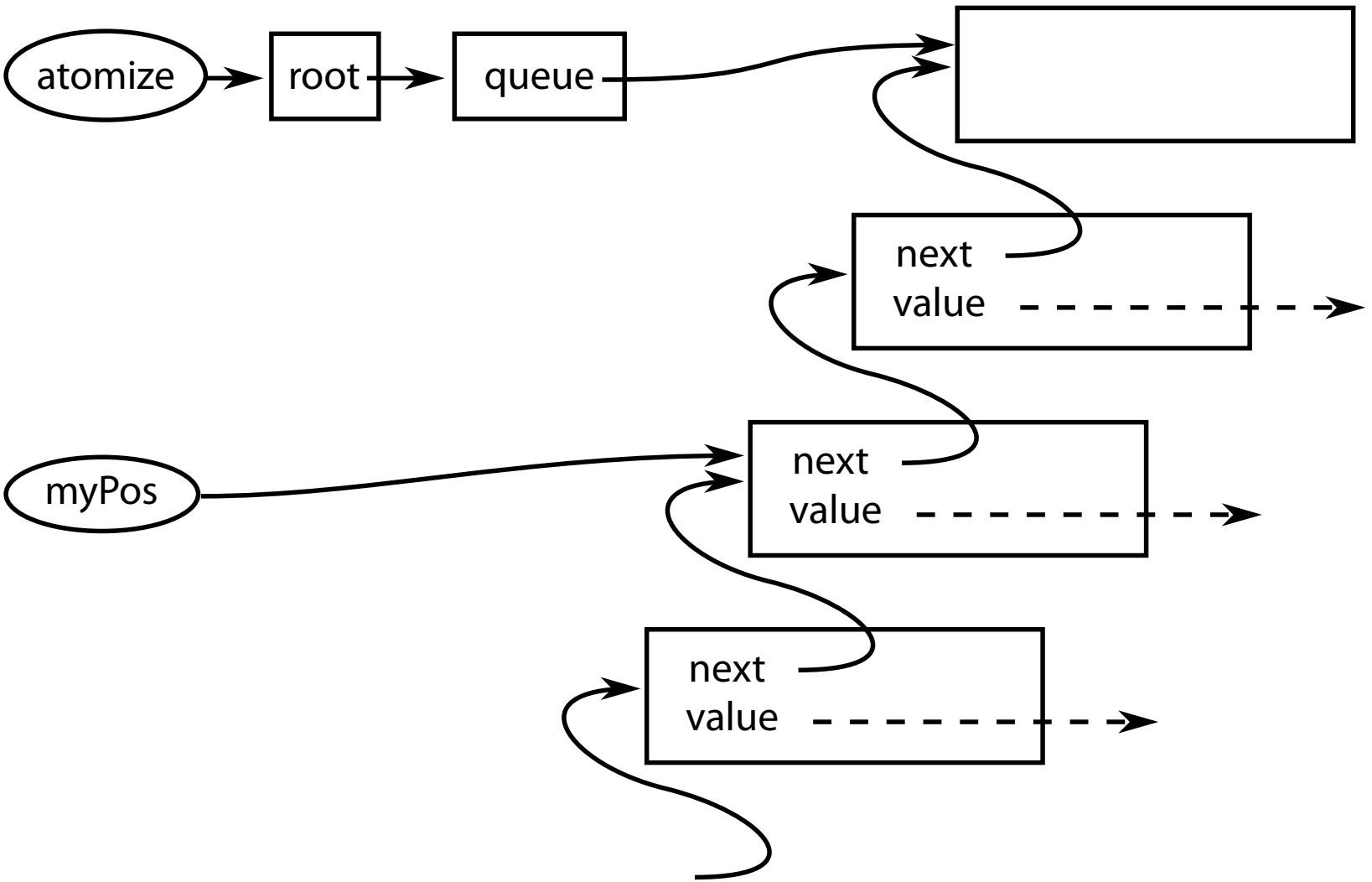
- Equivalent to very fine grained readers-and-writers locks
- Essentially: capture the effect of this transaction locally, and then take a read-lock on everything I read, and a write lock on everything I wrote to, and if after all of that I read and wrote to the same versions of objects as I've just locked then write the changes out globally, and release all the locks
- Opportunistic concurrency

AtomizeJS

AtomizeJS

- Server written for NodeJS
- Client library for both browsers and NodeJS
- A globally distributed `root` object against which you perform transactions
- Objects can be detached from the `root` object but are still managed by AtomizeJS

Example: shared queue



Example: shared queue - writing

```
function enqueue(elem, cont) {
    atomize.atomically(function () {
        var obj = atomize.root.queue;
        obj.next = atomize.lift({});
        obj.value = atomize.lift(elem);
        atomize.root.queue = obj.next;
    }, function (_result) {
        cont();
    });
}
```

Example: shared queue - reading

```
var myPos = atomize.root.queue;

function dequeue(cont) {
    atomize.atomically(function () {
        if (! 'value' in myPos) {
            atomize.retry();
        }
        var result = myPos.value;
        myPos = myPos.next;
        return result;
    }, cont);
}
```

Properties of shared queue

- No server-side code
- Anyone can safely write to the queue: concurrent writes will not overwrite each other
- Anyone can read from the queue
- Every client can read from the queue at their own pace
- Implementation is a plain, simple, linked list
- Actions on the list are obvious implementations, just wrapped in `atomically` calls
- Single writer and multiple readers will never cause conflicts at commit
- Use of `retry` means readers who catch up with the writers can immediately be informed when a new value is appended to the queue

Example: mongo-backed objects

Idea

- In the NodeJS server, watch for any changes so specific objects, and mirror those changes in Mongo
- On start-up, read data in from Mongo and populate the Atomize distributed object
- Clients can then read and write from/to Mongo by manipulating the normal object graph
- No RPC - no special APIs
- Make use of `watch`: wrapper around `retry` which calculates exactly what has changed since we last saw the object: map from object to object with 3 fields: `added`, `modified` and `deleted`, all lists of field names

Example: MBO - populating collections

```
populateAtomizeAndWatch: function () {
  var self = this;
  this.collMongo.find({}, function (err, cursor) {
    cursor.toArray(function (err, items) {
      atomizeClient.atomically(function () {
        var idx, item, itemCopy, itemAtomize, key;
        for (idx = 0; idx < items.length; idx += 1) {
          item = items[idx]; key = item._name;
          itemCopy = cereal.parse(cereal.stringify(item));
          self.collAtomize[key] = atomizeClient.lift(itemCopy);
        }
      }, function () {
        for (idx = 0; idx < items.length; idx += 1) {
          item = items[idx]; key = item._name;
          self.items[key] =
            new Item(self.collMongo, self.collAtomize[key]);
          self.items[key].watch();
        }
        self.watch();
      });
    });
  });
```

Example: MBO - watching collections

```
watchFun: function (inTxn, deltas) {
  var collDelta = deltas.get(this.collAtomize), idx, key;
  if (inTxn) {
    while (collDelta.modified.length > 0) {
      key = collDelta.modified.pop();
      collDelta.deleted.push(key);
      collDelta.added.push(key);
    }
    for (idx = 0; idx < collDelta.added.length; idx += 1) {
      key = collDelta.added[idx],
      this.addItem(key, this.collAtomize[key], true);
    }
  } else {
    while (collDelta.deleted.length > 0) {
      key = collDelta.deleted.pop();
      this.items[key].running = false;
      delete this.items[key];
      this.collMongo.remove({_name: key});
    }
    while (collDelta.added.length > 0) {
      key = collDelta.added.pop();
      this.addItem(key, this.collAtomize[key], false);
    }
  }
```

Observer-pattern and safely modifying shared state gets you a long way

- Grid, player locations and bombs are shared state
- Every time a player moves is 1 transaction
- Players responsible for detecting their own death
- Bombs are exploded by the player that planted the bomb
- Uses HTML Canvas
- Slightly latency sensitive!

Conclusions

Conclusions

- Simple and consistent paradigm
- Small but powerful API
- Easy and intuitive how to build richer libraries: both explicit communication patterns and shared data-structures
- `retry` supports trend for popular Functional Reactive Programming style popularised by Flapjax, Knockout, Meteor live-ui
- Ability to write identical code on client and server
- Fairly easy to make the server relay changes to other systems - i.e. act as proxy

Road-map

- Better story needed for older browsers: translation tool exists and works very well, but could integrate dynamically into NodeJS if NodeJS is serving static artefacts
- Security model: genuinely hard to work out what's wanted here
- Tutorials, demos, screen-casts, attracting audiences, making it seem less like hard-core CS!

Getting AtomizeJS

- `http://atomizejs.github.com/`
- Open source: MIT license
- `mailto:matthew@rabbitmq.com`

Thank you. (More) questions?