Federated working sets

David Barrett-Lennard
9 May 2013

Introduction

We will create a library called cxFederate whose purpose is to support federated working sets. Note that it is straightforward to create a single virtual tree by merging UTs at the position of the UTRoot.

cxFederate will encompass the following functionality

- Allow for a client server architecture
- Both clients and servers only hold a subset of all working sets
- Provide identity for working sets
- Support users and groups and access control lists
- Support servers having X509 certificates and secure socket connections - both between servers and between client and server
- Support user accounts with username, password and other data for each user
- Somehow cxFederate is extensible - e.g. for the data recorded on a given user
- Indexing of the working sets using extensible metadata

Load balancing and topology notes

Let *user-data* refer to the data viewed and edited by end users and *system-data* refer to other data used to manage the system and index the user-data. All user-data across the system is split into thousands (or millions) of *user-working-sets*, which can be downloaded and synchronised independently.

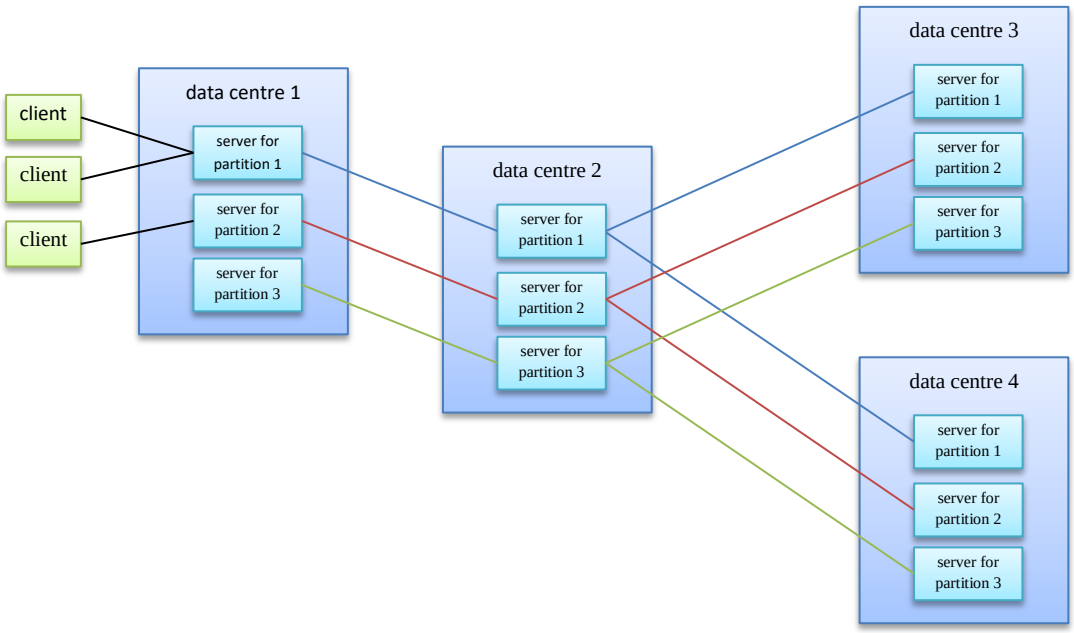
Let the system consist of a set of *back-end-servers*, accessed either by thin or thick *end-user-clients.* We call the client a thick client if it holds a local copy of a given working set during a viewing/editing session. It is assumed clients never connect directly to each other. Instead all communication is between client and server. Clients do not need to listen on ports and accept inbound connections, so there is no need for port forwarding. Servers provide reliable access to the data (if a client disconnects then other clients are unaffected).

In a very large system we cannot expect a single *back-end-server* to be capable of storing and synchronising every user-working-set. So the set of user-working-sets is divided into mutually exclusive groups called *partitions.* A user-working-set is assigned a partition at the point of creation.

Each back-end-server manages a single partition. For redundancy there can be multiple back-end-servers that mirror the one partition, typically at different locations.

A *data-centre* refers to a set of servers in a location. For the purposes of synchronising each partition let the servers of that partition be connected in a minimum spanning tree. On each

connection between them the *deltas* used to synchronise the working sets in the partition are multiplexed using a *working-set-identifier*.



Independently of this, all the servers are connected in a system wide minimum spanning tree for the purposes of replicating one of more working sets which contain system-data.

User information

Examples can be:

- UserId
- email
- username
- password
- avatar
- billing information
- groups that the user belongs to
- location
- Mail box (for messages that have been sent to the user)

We want to keep the basic system simple, so we will record the minimum amount of information possible, and assume additional data about users is recorded separately. This is easy if we have a stable UserId.

Groups and access control

A *group* is a named set of users defined by an expression. The expression can refer to users and groups by name and also supports set theoretic operators:

- + (for union)
- & (for intersection)
- - (for difference)

Note that changing the value of one group can implicitly change the values of other groups which reference it by name.

For example, the value of group G2 defined by

G2 = G1 + fred

depends on the value of group G1.

Grammar

todo: make the grammar define the precedence and associativity of the +, -,& infix operators. +,- have equal precedence that is lower than &. All operators are left associative.

```
userName = identifier;

groupName = identifier;

groupExpr  =
      'all'|
      'none' |
      groupName |
      userName |
      '(', groupExpr, ')' |
      '[', objectExpr, ']' |
      groupExpr, '+', groupExpr |
      groupExpr, '&', groupExpr |
      groupExpr, '-', groupExpr;
```

Example

G1 = bill+fred+mary

G2 = G1+burt-bill+barney

G3 = amy+wilma

G4 = fred+wilma

G5 = (G1+G2)&G4 - G3

Allowing for specialised models for ACLs

A specialised model for an ACL may be desirable for the GUI editing experience or for the way it allows concurrent data entry and robust merging. To allow this, consider that the grammar supports a reference to an object which implements an IUserSet interface for returning a set of users (or member testing).

Example:

G6 = [a.b.c] + G5

The path in square brackets identifies an object that implements the IUserSet interface.

ACLs

Definition: an *access mode* is a kind of access permitted on an object. Examples are

- all (a)
- own (o)
- read (r)
- write (w)
- execute (x)
- insert (i)
- erase (e)
- delete (d)

Using a letter for each access mode makes it possible to use a string to convenient define a set of access modes. E.g. "rw" for read+write access modes.

An ACL is represented in a single string, and defines sets of users with access rights. For example, the following grants read access to fred and read+write access to wilma and barney:

```
r:fred;rw:wilma+barney;
```

If access modes are not specified (but rather just an expression defining a set of users) then it is assumed those users have all available forms of access.

Distinction between username and groupname is opaque to other users

A name like shakespeare may be regarded as a username but may in fact be a groupname that passes on its access rights to multiple users. Other users in the system cannot tell the difference.

Microsoft call this a Trustee.

No root user

It is undesirable to allow administrators of the system to have access to user data. So there is no concept of a root user which has access to all data in the system.

What does ownership mean in a client-server architecture?

In a true peer system normally one would say a user only owns data on their own machine. But in a client-server architecture we have the concept of data on a server which is owned by a user. Let this data exist in a working set. The user will want to define the ACL on this working set. The user may expect this working set to be replicated onto other servers, but nevertheless the user owns all copies, and they all share the same ACL.

Scenario: User creates a meeting room with initial data. A list of attendees is defined. By default this matches the set of invites and the ACL. No-one else can edit the ACL.

Concepts

- Allow a user to create a working set and define ACL on it. User expects working set to be replicated eagerly to all relevant servers.
- A working set is assigned to a partition somehow at the time it is created.
- Publishing the meeting can mean adding it to an index of some sort. This is optional (maybe the meeting should be private)
- The system can provide a special user called "system" which is used to publish data. This is application defined. To cxFederated this is just another user, with working sets and ACLs. However, such working sets are not normally downloaded onto client machines. Instead they are only accessed through remote method invocation to services implemented on the servers.
- Sending of invites is an optional concept

Do we allow working sets to form hierarchies in some sense? What does this even mean?

Most data should exist in a working set

In general we want to put as much data in a working set, so it can be replicated. In particular it probably makes sense for each user to have a root working set which is like the home area. This is automatically replicated within its partition.

Trusting the servers

Each server has an X509 certificate and this provides the basis for trusting the identity of the machine to which a client has connected. Clients assume they can trust the backbone of servers to manage their data fairly and correctly and limit access to other users as required.

A malicious user must not be able to compromise a server.

Name server(s)

The name servers(s) are processes running on server machines that provide serialised access to a global namespace which is used to register globally unique usernames, groupnames and wsnames. The name servers manage mutually exclusive parts of the namespace, providing a *Single Source of Truth (SSOT)*. Use of OT is inapplicable to the name servers. A name server is accessed through remote method invocation. A name server serialises access to its state with a PSpace mutex.

In general an object never stores the ACL that defines access to itself. Otherwise there is potentially an irreversible situation where an object denies access to itself for all users. In particular a directory records both the names and ACLs of its child elements, but not its own name or ACL.

The access rights on a directory apply to its state which encompasses the names and ACLs it defines on its child elements but not the child elements themselves (obviously, because they have their own completely independent ACLs).

directory state = map<Name, pair<Acl, Element*>>

The access rights on a directory object are:

- (r)ead : permit read access to all the directory's state
- (w)rite : permit write access to all the directory's state
- (i)nsert : permit insertion of elements into the directory
- (e)rase : permit erase of elements from the directory

There are access rights on the children which affect what operations on the directory are supported as well:

- (d)elete : allow for object to be deleted from its containing directory.

A principle of orthogonality: There are different things to protect here and we must not confuse them:

1. The namespace system. This is all about protecting the allocation of names
2. The objects referenced by the namespace system.

What we have above is inadequate! The owner of a directory has nothing to do with the owner of the objects in the directory. That suggests we need two ACLs per object. The first defines who owns the object. The second defines who can access the object.

Access rights on an object owner:

- (r)ead : permit read access to the ACL of the object
- (w)rite : permit write access to the ACL of the object
- (d)elete : permit deletion of the object
- (n)ame : permit access to name of object?

IDEA: Maybe for each child object a directory records

- The name of the object
- an immutable string representing the owner of the object
- The ACL defining access to the object

Note that the owners have the right to edit the ACL of the object and to delete the object.

IDEA: Rather than use the username for the owner can we instead use a role?

Proposal: We want to avoid having so much data which is centralised and not getting the feature of replication and sychronisation. We need to realise that data on the replicated servers is still something which users may trust - maybe because it is only accessed by users which they trust. So there's no reason not to consider working sets to be a basis for providing namespaces, and having editable ACLs etc. We should minimise the centralisation.

Note as well that we can easily support two kinds of working sets

1. working sets which can be downloaded onto clients machines
2. working sets which only exist on servers and are protected by RMI. So they provide a weaker SSOT at quiescence, and even a strong SSOT if we talk about one server in particular.
3. consider a replicated map keyed by string. doesn't this map each string to a unique value? So even if we have duplicates they don't last long...

It is important to understand that a user doesn't need read access to a directory in order to be able to bind to elements under it. The ability to bind to an object only depends on the final ACL, not on the ACLs along the path. This allows otherwise private areas to publish things.

Definition: A *systemspace* is a namespace identified by a path which is managed by a single name server. For example the namespace root (/) is a systemspace. Each systemspace has an owner identified by a username and records the following information:

- an ACL which defines the set of users which have permission to create system or user namespaces under that systemspace on a first come first served basis.
- a set of named userspaces. For each userspace the owner is recorded
- a set of named systemspaces. For each systemspace the owner is recorded

Note that only the owner of a systemspace has write access to all its state - and therefore is given the right to:

- edit the ACL
- delete existing registrations
- edit the owner of existing registrations.

The root *systemspace* is implicitly owned by the username *rootsystemspaceowner*.

Definition: A *userspace* is a namespace identified by a path which is allocated to a single user. The user can create names as required under a userspace without registering them with a nameserver (indeed attempting to register names under a userspace will fail).

```
typedef string Name;
typedef string Username;
typedef Username Owner;

$variant NamedEntity
{
   { Owner }  UserSpace;
   { Owner, SystemSpace }  SystemSpace ;
   void User;
   void Group;
};

$model SystemSpace
{
```

```
    Acl CreateAccess;
    map< Name, NamedEntity>  NamedEntitys;

    //map< Name, Owner>  UserSpaces;
    //map< Name, pair<SystemSpace, Owner >  SystemSpaces;
};
```

Domain server

There is a single *machine* in the system which is responsible for issuing root domain names. Centralisation is just to ensure that domain names are unique under /.

In general a given domain path is mapped to a machine which is responsible for administering that namespace - to ensure that all names registered in that namespace are unique. The machine is accessed through RMI. The process will typically serialise access to data with a PSpace mutex.

Most generally a *domain server* takes on responsibility for administering the following names in its namespace:

- usernames and groupnames
- workingsetnames
- domainnames

[idea: do we need to generalise this concept - so we have a generic concept of registering servers which provide a service - through remote method invocation]

It may be appropriate to dedicate a process for a domain server. This helps ensure the process doesn't crash - because it is doing very little and therefore is unlikely to have a bug. It is a good idea for central servers to be as resilient as possible.

Users which own a domain

A user can own any number of domains for naming all their working sets. In particular a username is always available as a domain name.

E.g. ~/jigsaw1, where ~ denotes their home domain.

This eliminates the need to verify a name every time they want to name something. It also avoids polluting the global namespace.

Idea: one namespace to suit them all

Consider that usernames, groupnames, wsnames, domainnames all exist in a single namespace, with '/' for separators. Example

/microsoft

/microsoft/steve.ballmer

/microsoft/steve.ballmer/jigsaw1

Every node of the tree has an ACL on it.

Trusting user identity

We do not allow duplicate usernames in the system. Usernames/groupnames are meant to be unique, immutable and human readable to make it easier to validate ACLs. A malicious user cannot masquerade as another user because duplicate usernames are impossible. It follows that usernames are protected like company names and trademarks, and indeed a user may consider their username to be very valuable to them.

If a user wants to change their username (e.g. Mary Smith gets married) then the user should create an alias. One way is to turn the old username into a groupname that forwards on its access rights to the new username.

The ability for a user to have multiple aliases is useful and important. For example the alias email/joe.smith@xyz implies they formed the account after receiving a confirmation email at the given address. The alias acme/joe.smith implies they work for Acme Incorporated.

A concept of domains is helpful for allowing usernames or groupnames to be trusted. It also provides the basis for decentralised allocation of usernames and groupnames.

Trusting data identity

A working set is replicated on all servers of a partition. As far as the users are concerned there is only one distributed working set.

All working sets in the system have a unique identity.

We expect working sets to cross reference each other by name - just like the way web pages cross reference each other using URLs.

Therefore we assume working set names are

- human readable
- globally unique
- decentralised using domains
- immutable
- support aliases

Mapping username to partition index

Consider there are a billion users with 100 billion working sets, and 1 million partitions. Let there be 100 bytes of data per user - so 100GB. Is it reasonable to index all the users in a single machine? Using up 100GB on every server seems like a bad idea.

Consider that a partition manages a subset of users. This is based on a hash on the username. So given a username we can calculate which partition manages that user and go straight to that partition. That's great for performance. But how do we increase the number of servers over time?

We can overcome this problem by allowing a client to go to any partition and be redirected to another one. In such cases a partition simply

1. calculates which partition is appropriate for the given user, by calculating a hash on the username.
2. finds the IP of the local server for that partition
3. returns the IP in a message saying "please connect to this machine instead"

What happens when more partitions are added? Servers must be added at every data centre. Then somehow we need to atomically(?) turn them on so they can start managing users. This raises a number of questions.

Let there be a hash function on username that returns a 20 bit number (noting that 2^20 is about a million which is the maximum number of partitions). Let each partition record something analogous to a routing table. This is a look up table with 2^20 elements which maps each hash value to a partition index. On this basis, any server can easily map a given username to a partition that has been allocated that username to manage.

This table provides tremendous flexibility for adding new partitions to the system and incrementally offloading user accounts to those partitions. Note that partitions need to support transfer of user data to support a changeover. We can also support a period where more than one partition may potentially store data for the user, in some kind of priority order. A partition stores a parent partition index from which it was derived. If a lookup fails it is redirected to the parent.

To reduce overheads when the system is small, consider that only the highest N bits of the 20 bits are used as appropriate. This allows for a much smaller LUT. It doubles in size when it grows. When it grows each entry bifurcates in two. Then entries can be assigned to new machines as required.

A partition can use the LUT as a basis for automatically sending data to the appropriate destination.

The same thing can be done for a working set name (i.e. it is mapped to a partition using a hash function).

PartitionWorkingSet

Consider a working set owned by some user and replicated on some partition. This has an ACL which defines which users have read access and which users have write access. The ACL is replicated. The ACL is not stored in the working set itself.

Each partition has a PartitionWorkingSet which is replicated on all servers in the partition, so at quiescence they agree on the information recorded in the partition.

A partition manages a bunch of working sets, uniquely identified by wsname (working set name). These are indexed by each server in a single map in the PartitionWorkingSet.

The map records three strings for each entry:

- wsname
- ACL

- owner (as a username or groupname)

Problem: Given that we don't have atomicity between a partition and a name server, how do we reliably add a working set with a given name? What if two users try to add a working set with the same name?

It is proposed that the following steps are used

1. First go to the domain server and try to register a new working set name by the given username. This returns either
    a. success - it was created
    b. failed - a different user has created a working set with that name
    c. already created - that user has already created a working set with that name
2. If a or c then proceed to assign an entry into the WorkingSetIndex of the appropriate partition. Note that this may assign over the top of an existing entry but that's ok.

Name conventions

Basic usernames

If a new user wants to create an account themselves, then obviously they don't have any special access rights, so they can't give themselves a name with a format that is reserved for members of groups. Nevertheless we might allow a billion users to create accounts with any name they like as long as they are unique case insensitive alphanumeric identifiers of a minimum size. Note for example that spaces and characters like ~ ` ; : ' " = ! @ # $ % ^ + - & ( ) [ ] { } * / \ ? < > . , are not permitted.

Even with only 6 characters there are over 2 billion strings, so in theory users will be able to create reasonably short usernames. If there are many users then they may need suggestions for picking a unique username.

Users shouldn't be too concerned with the elegance of their username, because they can introduce aliases or forward addresses. They can even use aliases for logging in to the system.

Groupnames

There are other namespaces besides the one for the basic usernames.

Public interfaces

Rather than think about how to implement the system we should start with the public APIs. Let's ignore the fact that the system is distributed for the moment, and think about the functionality available to a given user, accessed from their client machine. Functions are:

- Create a new account.
- Log in / out
- Create working set

Creating an account

Creating an account requires a new unique username to be provided. If it matches an existing username then it is rejected.

However we want to promote the idea that group membership can change over time, so we think of the special format names as aliases or "forwarding addresses".

We don't want to pollute the global namespace with a billion users which will create all manner of silly names. Therefore these names exist in a namespace prefixed with #. E.g. #pete102. The # is a reminder that these names are a free for all. The # is required when the name appears in an ACL. However it is not needed when pete102 creates his account or provides a username to login.

Actually, rather than fiddle with the public namespace, lets instead reserve a private one.

We want to allow a user to create an account and later the user is added to various groups. In the process aliases of the name become possible.

26^8 = 200 billion.

How can a user add themselves to a private domain? Perhaps a user who is logged in creates an account on behalf of another user?

We seem to have two concepts that overlap in purpose

- domainnames, used to provide namespaces for usernames that can be trusted
- groupnames

So let's merge the two concepts. Being a member of a group entitles one to qualify the name with that group. E.g. providing access to microsoft & steve.ballmer gives confidence to users that are trust the validity of the group named 'microsoft'.

A group is essentially a predicate on the set of users.

Actually, why not allow a user to define their displayname as they like and applications can easily find what groups they belong to.

Idea: consider that usernames do not have domains in them, and we don't allow usernames to have '/' characters in them. We expect users to create an account themselves. If they want to join a group then they need permission from the group owner. The effect is to give them an alias.

Sometimes ??? HmmThis requires a new unique username to be provided. If it matches an existing username or an attempt is made to add it to a namespace for which the user does have permissions it is rejected.

Maximising use of replicated working sets on the servers

Working sets on the servers can diverge so don't represent a SSOT. However we can treat one as special and call it the SSOT. Nevertheless it can be edited remotely as it were.

Working sets which are edited in real-time don't give rise to divergent tasks in the minds of the users, so presumably it is unlikely that significant conflicts will occur.

Consider that a working set provides a library for "child" working sets. E.g. it may index a whole lot of jigsaws. Then we want the ACL information to be available for collaborative data entry according to access rights. The library can use a map from string to give a trustworthy notion of named objects.

todo

- We can prototype quickly using simple structs, all in process. We can unit test name servers and help get the basis ideas right
- the basic namespace for users can be implemented on a single machine - even for a billion users
- the basic namespace for unqualified names should be mounted in the global namespace tree under some reserved name.
- ebay users could be mounted under a directory called ebay
- emails could be mounted under a directory called email
- internet domain names under a directory called www
- forget about "owners". Instead just have ACLs in the namespace tree. ACL is much more flexible because can talk about read/write/create/delete rights etc.

Tasks

1. Create library cxFederated. Use boost, openssl, cxOperation etc
2. allow boost.asio to be used with cxOperation and cxRmi
3. allow secure sockets to be used
4. Design the authentication system. So have concept of trusted names of users, groups, working sets. Use proper models for all this so all information exists in a working set, allowing it to be replicated easily.
5. Design the authorisation system
6. Implement a partition server
7.