System Call Proxy Protocol

January 3, 2026

1 Introduction

This document describes a distributed dataplane to support clustered operating system functions. For no strong reason we have selected triples, or entity-attribte-value(EAV) assertions as the foundation. Arbitrary length tuples with schema, basically relational tables is another suitable substrate, as is a simple nested hierarchy.

We put a little structure around our triples and then address some of the issues around distributed evaluation of programs that operate on this EAV graph.

2 Triples Model

Triples came out of the semantic web, in a specification called RDF, where unfrotuantely the mapping between XML and RDF was left undefined (). Objects in the system are given globally unique identifiers. Each of these objects has a set of properties described by the attribute field, and each attribute for an entity has exactly one value. So we can view EAVs as really just a property list or hashmap.

Since the value can be another entity, triples can also be used to form a graph of entities with labelled edges. When developing schema for this system (for some reason we use the word ontology), its quite common for the attributes themselves to be objects, although this isn't a usage we will pursue.

The objects of interest in our system are things like:

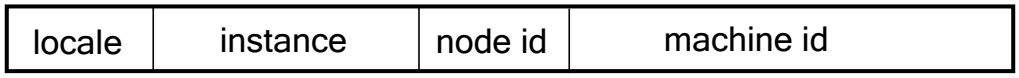
- machines
- processes
- users
- files

It shouldn't be contentious that we can use this graph to model operating system state, in particular the posix-esque semantics of linux syscalls. If not, hopefully the following details will paint a better picture.

2.1 Entities

Entities are the objects of our system, and to facilitate distribute evaluation we will also use them as addresses for purposes of routing. We also need to construct them in such a way that they can efficiently generated, and that we can never have two objects anywhere in the world with the same name.

There are probably too many dependencies on the ultimate distribution methodology to make an authoratative design, but this can serve as an illustrative example:



2.2 Attributes

In many triple setups, the attributes are entities themselves. This allows attribute metadata such as kind of modifier, historical information, references into specifications, etc. At the other end of the spectrum attributes are selected from a small set of predefined integers. For the initial system we propose a middle ground, that attributes be freeform strings. In order to bring some modicum of standardization and control to the layout of objects, we introduce a simple schema definition to publish known keys and support validation.

2.3 Values

3 Schema

The invariable endpoint of an ad-hoc embedding (like unix into triples) is a kind of loss of design cohesion. Some parts of the shape are deprecated but perhaps some usages are still maintained for convenience. For some reason one ends up with two canonical idenfiers for the same logical object and has to choose between several ugly alternatives to map between them. Objects are given multiple oids, etc.

One way of exerting some control on this evolution is to introduce schema for the data. This at least attempts to describe the shapes that are valid, serves as a form documentation, allows incompatibilities in usage to be caught at compile time, or startup time, and lots of other structural benefits. It would also be lovely if we could include additional metadata in the schema, for example allowing tools to know that they can join through a particular attribute to get a useful printable name for an object.

Since everything is stored and communicated as triples, the schema should be reflexive, that is objects are instances of schema, which are objects that are instances of the metaschema, which is a single schea-object. [I recently saw reference to a paper that shows that this arrangement leads to decidability problems. i should really find and read that].

Unfortunately this means designing a small type system, expressing it in itself, and writing it up here. The primary usage will be the declaration of objects, meaning their attributes and the types that the corresponding values can be elements of. Certainly need to support optional fields.

Additionally we would like the attribute to be any member of some type, where all of their values are of the same type (i.e. Hash¡K,V¿). This is used in directories `(String -> {File  | Directory})`, and extent maps including virtual memory translations `(u64 -> Extent)`.

4 Commands

4.1 set(entity:Oid, attribute:Value, value:Value, status:Object)

4.2 get(entity:Oid, attribute:VAlue, value:Value, result:Variable)

status here?

4.3 copy(dest:Address, source:Address, length:usize, status:Object)

4.4 create(dest:Variable, source, length)

Copies *length* bytes from the *source* bytestring to the *dest*. It is an error if both the referred objects are not byte strings. For convenience if the destination is an (entity, attribute) pair which current is unbound, then the binding is created and set to the bytestring specified by the source address.

As a convenience function, an address which points to an (E, A) that is empty instantiates the binding with a byte string value of the length. This likely shouldn't be implicit, but require a distinct command.

4.5 Addresses

The *source* and *dest* parameters are compound addresses. Addresses refer to byte ranges in values which must be byte strings.

value(e, a)

offset(A, count)

Originally addresses were conceived as some kind of nested routing infrastructure, but that whole line of thought missed the point that entity is does the reachability work here. There was also a range instead of an offset, but since that implied a length, that admits a case where the source and destination are mismatched and must throw an error.

The primary resason that they still live on in the design is the possibility of using them to express various address translation mechanisms as rewrites as address terms get evaluated. The most interst example of this would be the translation of a process VMA to a RDMA window+offset. This along with some RDMA completion plumbing would enable third party transfers for distributed interleaved requests.

5 Program Blocks

Collections of commands are grouped together into *blocks*. One reason for doing this is to avoid alot of request/respose trips when performing compound operations such as path resolution.

One of the reasons why we defined a command batch was to try to promote some notion of atomicity or transactionality. Actually achieving this without substantial investment in the host stack is likely impossible. In general a systems operation (like writing a block from a storage device) cant be validated completely in advance. Nor is it likely that generalized undo strategies for many of these operations can be developed.

Regardless, there is incremental value in issuing multiple related updates in a single operation. Batches allow us to bundle multiple operations to avoid round trips, and thus allow to use a fine-grained schema which would otherwise be costly. Even if we cant promise a completely atomic operation, for operations like object contruction we can validate the entire set of initialization values to ensure that they are consistent. Furthermore, we avoid problematic states where the object is partially initialized.

5.1 Evaluation

The block evaluation has semantics designed to make it simple to support multiple results as streams, at the expensive of having to explain them more carefully.

Take a command in our block, it takes some arguments. In order for these to be chained and permit other interesting uses, we allow some of these terms to be variables bound at runtime. So we can express a general copy from A to B where variables are denoted with a '

```
get A attr %1
set B attr %1
```

We generalize the binding of variables to include multiple values. In logic we would say that variables are universally qualified rather than existentially. In programming we would say that every command contains an implicit `forall` loop around it. Variables are all initialized to to the universe of all values (U or $\top$).

At each step in the evalution, constants are treated as unit sets, and for variables we take the intersection of the current value with the valid values that that variable can take in that expression. This results in a naive implementation of unification. The most trivial implmenetation of multiple results is to simply evaluate the succeeding expressions for every combination in the intersection.

So now we can copy all the attribute/value pairs from A to B by allowing `range over all valid entries.`

```
get A %1 %2
set B %1 %2
```

`We would also define operations on variables` to be the union rather than the intersection in some cases. The collection of a set errors is notable one. Its not clear if this should be by command slot, or some additional dynamic notation in the application, or...

5.2 Results/Continuations

We've painted a picture of an evaluation tree, where we can split work if its worthwhile to do so, but we haven't discussed how results and errors get propagated back to the original caller. Since we're not going to really be programming very much of this stuff, lets avoid adding more constructs and just say that

5.3 Error handling

Three of the four (set, get, copy) commands take a result or status. Exactly how this gets expressed is somewhat up to the evolution of the type system. The intent is to have Error objects with fairly substantial metadata, and some convenince functions to generate them. This is necessary in a very decentralized distributed system in order to identify which software compoents failed, and on what machines, and importantly to correlate the error with a particular distributed evaluation. By the same rationale, the need to be structural because they will be processed by software.

Now that we've imposed alot of semantics onto the error, we can also exploit them to good end. It should be possible to enable the embedding of timing and other performance information into status/error objects. Oids which identify a particular process can be later used to run queries on things like age, binary provenance, memory utilization, identifiers for attached storage hardware, etc.

The syscall proxy and posix services need to agree on a canonical errno description, be prepared to attempt to map the local environment to the canonical, and include that information explicitly in status objects relayed as part of a syscall.

6 Concurrency

The current implementation does not provide any global isolation or serializability for the execution of blocks. If services are provided on top of a system like Linux, then its very difficult to introduce these concepts because the underlying facilites aren't 2-phase - they cant in any way promise to commit.

7 Security

There is a nice design where blocks are signed by principals, and contain signed statements of delegated authority. A user might start a process and sign a statement including the processes new public key and a set of grants in EAV space.

If we insist that programs can be rewritten by intermediate nodes (for example the interleaved storage example), then we are opening the door for privledge escalation, because intermediate nodes must be able to have very permissive grants to handle any clients traffic, and if the policy of the intermediate and ultimate nodes about the principal differ, then we can easily violate the policy of the ultimate node.