UUIDs

Wei Hu

1.0 Introduction

UUIDs (Univeral Unique Identifiers) are ubiquitous throughout the file system. Almost all the objects of interest (volumes, files, queues, etc.) have UUIDs. This document describes the interfaces to the UUID module and how UUIDs are generated.

UUIDs should be unique and cheap to generate. UUIDs will be stored on disk as part of our file system metadata. Since we expect to eventually run xFS in a heterogeneous, distributed environment, we should expect that UUIDs will be transmitted over the wire and stored in name services. This also means that UUIDs must be extensible to non-SGI environments.

UUIDs will be opaque. We do not expect to derive, for example, the node that generated a UUID.

This design takes the approach that we should adopt the OSF/DCE (which is the same as the X/Open) UUID mechanism and make changes only where the existing mechanisms do not work. I will further argue that we can in fact adopt the DCE UUID mechanism as it currently stands. However, 2 alternatives are also presented as fallbacks.

1.1 Interface

A UUID is a 128 bit quantity whose internal structure is not exposed. The following are the operations that can be performed on UUIDs:

- uuid_create - create a new UUID
- uuid_create_nil - create a nil UUID. A nil UUID is a distinguished UUID whose bits are all 0. For debugging purposes, we will redefine the nil UUID to be some distinguished value and check to make sure that we never get an UUID containing all zeroes.
- uuid_to_string - converts a UUID from its internal format into a string format
- uuid_from_string - converts a UUID in string format into its internal format
- uuid_equal - compares 2 UUIDs
- uuid_is_nil - returns true if uuid is nil
- uuid_compare - lexically compares 2 UUIDs
- uuid_hash - returns an unsigned32 hashed value for a UUID

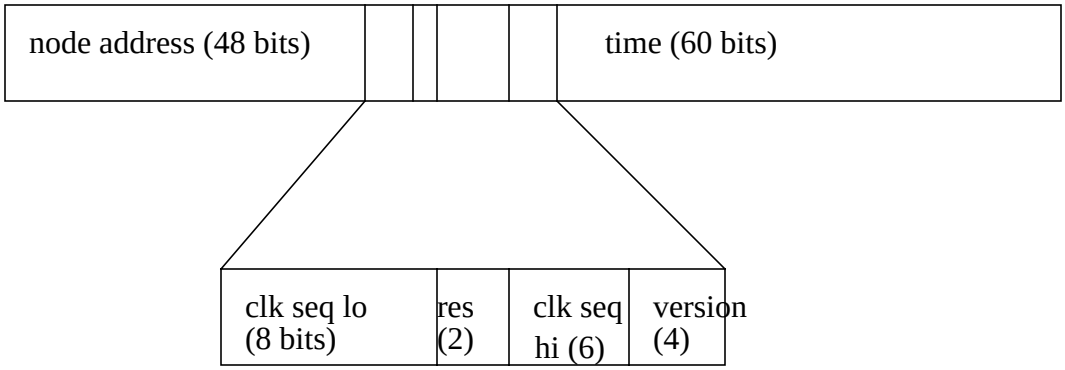
Note that these are exactly the UUID operations as defined by OSF/DCE. This set of routines appear to be adequate to our needs and we should adopt them as is.

1.2 DCE UUIDs

UUIDs derive their uniqueness from a space and a time component. The space component is typically some hardware-based serial number and the time component is derived from the current time. To understand how we will generate SGI UUIDs, we will first review what DCE UUIDs look like.

DCE knows about three variants of UUIDs. Variant 0 was the Apollo NCS UUID, Variant 1 is the DCE UUID, and Variant 2 is defined by Microsoft. The 3 MSB of octet 8 contains the variant. All of these UUIDs are 128 bits long and have the same record structure, this is significant as it allows the integer subfields of these UUIDs to be byteswapped the same way when transmitted over the network.

The following is the layout of DCE (variant 1) UUIDs:



The node address provides space uniqueness. It is the 48-bit IEEE 802 address.

The 60 bit time is the number of 100 nanosecond intervals since 15 October 1582 (date of Gregorian reform to the Christian calendar.). This is split into 3 fields (an unsigned32 and 2 unsigned16s). At 60 bits, the time field will roll-over at approximately 3400 AD.

The “res” (2 bit) field determines the type of the UUID (NCS, DCE, Microsoft, etc.). The clock sequence is used to augment the clock so that we would still generate unique timestamps even if we crashed or if the clock did not advance fast enough. The 4 bit version field captures the version of DCE UUID; currently there’s the DCE version and the DCE Security version that imbeds POSIX UIDs.

1.3 Generating a DCE UUID

The node address and time are obtained from the system by conventional means. Setting the clock sequence number is more involved:

Whenever the system is rebooted, we can either: 1) increment the clock sequence number last used if it's saved in nonvolatile storage, or 2) initialize it to a random number. We need to do this because we couldn't tell what the last time that was used to generate UUIDs. xFS will save the last sequence number used on disk.

Note that initializing the clock sequence number to a random value has the secondary benefit of guarding against duplicate node addresses.

In addition, when generating UUIDs, the clock sequence number must be incremented (modulo 16,384) if the UUID generator detects that time has gone backwards To do this, the UUID generator keeps the last time used to generate UUIDs in core.

To allow parallelism when there are multiple CPUs, we will allocate allocate a different sequence number to each CPU and then allow them to independently generate their own timestamps without having to lock a global datastructure.

If UUIDs are created faster than the clock resolution, then the system would just increment a counter and add that to the low order bits of the time. This counter would be cleared the next time that the clock advances. Since reading the clock is a fairly expensive operation, we will likely just increment the counter until we've consumed about a second's worth of bits.

1.4 DCE UUIDs are Sufficient

I believe that we should just use DCE UUIDs as defined, including the reliance upon the IEEE 802 address. This has several advantages:

- We don't reinvent the wheel.
- The algorithm has good performance. Based upon prior experience, the only bottleneck appears to be the gettimeofday() system call that gets the current time. We can reduce it further by incrementing the low order bits of the time value and only reading the system time when the counter is close to overflowing.
- We are compatible with DCE UUIDs. Thus, programs that display, compare, or transmit DCE UUIDs will be able to transparently handle our UUIDs also. For example, our UUIDs will be automatically byteswapped correctly when transmitted via DCE RPC.

The primary concern about DCE UUIDs has been that the IEEE 802 addresses are not unique. I have done a bit of investigation on this.

For low end SGI machines, the IEEE 802 numbers are now unique. ASD manufacturing keeps a database of numbers that have been assigned and rejects duplicates. There are a few ESD option boards which do not go through this process; I'm in the process of finding out what they do.

Note that because of the way sequence numbers are randomly initialized, even the existence of duplicate IEEE 802 addresses is unlikely to cause a problem.

Note that since we do not map UUIDs back to their IEEE 802 addresses, swapping Ethernet (or FDDI) boards across machines does not cause a problem for UUID generation.

1.5 Alternate Node Address

We had considered using the 32 bit sysinfo serial number as a basis for generating a node address. This does not appear to be a good alternative because on all platforms other than Everests, the sysinfo is derived from the IEEE 802 address. Furthermore, it's not clear that the chassis numbers (from which Everest sysid numbers are derived) are inherently any more unique than IEEE 802 addresses. Thus, I think we should just rely on the IEEE 802 address as is.

If we decide not to go with IEEE 802 addresses, then we must make sure that our UUIDs don't collide with DCE UUIDs. Otherwise, we might have an SGI UUID that has the same value as a DCE UUID and be very confused. (This might happen, for example, if we exported our volumes into a DCE directory service.) There are a couple of ways to do this:

1.5.1 Reserve a Block of IEEE Addresses

I have checked with the IEEE and confirmed that, for $1000, we can get our own block of IEEE 802 addresses. They come in 24 bit chunks. IEEE will give us a block of numbers by generating random numbers for the upper 24 bits. We are then free to assign the lower 24 bits as we like. Our node address would therefore consist of the 24 bit number that is uniquely assigned to SGI plus a low order 24 bits that's derived from the machine serial number. This will guarantee that SGI UUIDs will not collide with any DCE UUIDs.

This scheme has the drawback that we can only use 24 bits for our serial number. (24 bits is about 16 million.)

As stated earlier, any scheme that relies upon a machine serial number is currently only applicable to Everests.

1.5.2 Use Different Version Number

I do not advocate using this approach. It is captured here if we decide that any of the previous approaches are not workable.

The other way to differentiate SGI UUIDs from DCE UUIDs is to make SGI UUIDs a DCE UUID (i.e., variant 1 UUID) with a version number of '1100' (12 decimal). This is not one of the versions currently in use. We should also let OSF know that we are picking this value to prevent conflicts. Once we do this, we are free to assign all of the 48 bits in any manner we choose.

The 48 bit node address will be assigned as follows:

- 6 bits name domain identifier. We woudl split the world up into separate name domains and select one agency to distribute company identifiers for its domain. SGI, for example, might be the agent that distributes company identifiers for all the companies in California. Having multiple domains allows multiple agents to independently distribute identifers without conflict.

- 12 bits company identifier. This identifies the company - e.g., SGI. This gives us 4K companies within each naming authority. Note that locations with a high concentration of companies might be divided into multipl name domains.
- 30 bits vendor-specific identifier. How this field is assigned is up to the vendor. SGI might, for example, use machine serial number.

These fields are hierarchically assigned so that within each level, unique bits can be generated without conflict. SGI can, once it has its assigned 6 bit naming authority and 12 bit company identifier, generate unique 48-bit ‘node addresses’ just by assigning unique 30-bit numbers to its machines.

1.5.3 Compatibility

Making the SGI UUID a new version of the DCE UUID that uses an alternate means of assigning the 48 bit node address is the next best thing to using DCE UUIDs directly. We don’t reinvent the wheel, and SGI UUIDs can still be transparently handled by utilities that expect DCE UUIDs.

1.6 Issues

The only outstanding issue is compatibility with Lego UUIDs. Lego currently uses another UUID format and appears to use a subfield for object class information.