VEOS INTRO

dav lion
March 1991

The veos is a framework for designing networks of distributed communicating processes. VEOS is written in C on top of UNIX and TCP/IP, to provide some measure of platform independence. It currently runs on/between sparcs running SunOs and dec 5000's running Ultrix. Veos is everything and nothing, or perhaps it just seems that way to me, as i am much too close to it. This document is intended to be a tutorial of veos application ("entity") programming, with code examples and explanations.

The veos provides

message passing across machines	(the "talk" library)
a consistent data representation	(grouples)
foreign function calling	(primitives)

The veos DOES NOT provide

any notion of i/o devices	(spaceballs, keys, screens)
any notion of intended application	(virtual worlds, databases)
built in control structures	(no conditionals are provided)
sophisticated debugging facilities	(use dbx or gdb)

Known Problems and Limitations

using strings in C is a hassle, and slow. yeah, we know about lisp.

unix may not clear internet sockets immediately after processes die, thus, if one entity goes down without cleanly closing its socket, communications may be messed up for a while

The veos could be used for applications other than virtual worlds, and virtual worlds could certainly be written without the veos. We wrote the veos to provide ourselves a FRAMEWORK in which to write distributed virtual worlds. The veos is an operating system on top of unix. This means that the veos inherits all of unix's goofiness, including the sponginess of unix buffered i/o, especially network i/o. If the underlying unix can not guarantee hard time constraints, neither can the veos.

The veos is a development environment which provides the basic high level facilities enumerated above, so that the programmer can concentrate on the functionality and design of the application, without having to worry about low-level details such as inter-machine communications and remote function calls.

What VEOS is good for

1. VEOS is good for prototyping distributed applications

This is heterogeneous distribution, which means that your application may be split up to make the best use of mixed resources. For example, you could render graphics on a dec5000 that were computed on a set of Sun's.

Since distribution of processes is course grain parallelism, veos can help model parallel systems. In fact, from the start the VEOS has been intended to provide a loose parallel architecture.

The talk library provides fairly easy protocol and facilities for message passing between separate unix processes linked by Ethernet. If you can break down your application, you can immediately parallelize it, simply by connecting the pieces via talk. You need only pack your data into grouples at either end of the communication calls to conform to the protocol. That's it.

2. VEOS is good for prototyping parallel applications

I described the parallel architecture as loose, because the VEOS has avoided many hard problems, such as synchronization, actual shared memory, and load balancing. We know that these problems exist, but we did not set out to solve them. As the veos is a prototype environment, solutions to these problems could be implemented ON TOP of the VEOS.

Shared Memory can be simulated by having one entity maintain a data-space, and having all other entitles make use of it for their memory. This was used for flockworld, a distributed rule based group behavior simulation, in which separate entities kept their position externally in a geometric space entity. The space entity served as a shared database. In the same fashion, a Nancy server entity could be written, so that all grouple space was non-local. This is reference to the Linda database model which was influential on the veos design.

Synchronization can be overlaid by having all entities set semaphores with some common entity, or, optimistically, with a clock entity, which sends out ticks.

Entities can report how hard their cpu is working to a central load-balancer, which can redistribute workloads accordingly.

3. VEOS is good for dealing with multiple types of devices.

Well, as good as any arbitrary standard can be. Grouples may not be ideal (certainly not in C, i admit), but they are consistent. Device drivers, really grouplefiers, allow connection of arbitrary devices into a veos network. Remember, if all data is a grouple, and all entities use grouples, then all entities can use all data. Consistency may be the hobgobblin of little minds, but sometimes it's a decent tradeoff.

4. VEOS is good for C hacks who don't want to learn lisp

This is certainly a big reason why *this* version is in C.

The Model. Glue not included

Ah, the model, that bastion of modern thought. Nobody really understands anything, but they have a model for it. Well, the model is useful, and so here one is. (As i'm a sucker for an extended metaphor, . . .)

(For historical note, i must credit Espresso Roma for their fine fine patio and superb caffeination. We (geoff coco, dan pezely, bill gates, and myself, dav lion) spent many afternoons there, alienating those around us with our wired ranting of abstract architectures, while trying to figure out what was needed. William Bricken didn't accompany us to the Roma, alas, and is excused from errors induced by latte'. The physics metaphor is new (just for this document.)

PRIMITIVES are the atoms of veos. They are C functions which codify rules outside of the groupleSpace. This is keeping with the C paradigm of the separation of data and control-structure. Primitives are public functions. Primitives have a consistent data representation.

ENTITIES, as described earlier, are the molecules of veos applications. Every Entity has the following basic capabilities:

communication
inference engine
data-space

Communication is provided by the talk library, inference is minimal match and substitute, provided by the shell, and groupleSpace is the data-space.

In the model, the inference engine processes the data-space, applying all rules which apply to each piece of data. In reality, the shell processes the groupleSpace, passing grouples to local Primitives which post their results directly back to the grouple space.

The end result is the same, the grouples are interpreted; the difference is that all rules are not codified in the space, some are coded as C functions, which are called by the shell. This is how the shell can perform the minimal inference of match (on keywords) and substitute (Primitive function calls).

WORLDS are the crystalline structures of veos applications. They are made up of mixed Entities (molecules), ordered by some protocol. Just as crystals can grow, veos application networks can grow, and just as pieces of crystals can break off (go out and buy some rock candy right now, if you don't believe this part), pieces of worlds can break off. Crystals can react to external stimuli, and so can veos applications.

The danger of this metaphor, is that it implies a sort of static form, a rigid solid, which is not necessarily true for a veos application. To date, we have not built any fluid oozing worlds, which i now suppose is necessary. ah, the price of an extended metaphor.

What Grit Sandpaper to use?

What is the granularity of your application? When does it make sense to have a separate entity, instead of a separate primitive? How much data should be kept in the groupleSpace, and how much in a local data structure/cache?

Designing a world in veos is strange. We have built two that have lasted more than a week, flockworld and blockworld. (i suppose the next should be crockworld, a liquid world of questionable worth). What we learned building them boiled down to this:

> Early design without concern for real-time issues
> led to fast conceptual justification through
> working worlds, yet atrocious performance.

Hey -- this is one of the touted benefits of the veos, that you can start using it quickly, but, there ain't no free lunch, and if you want a fast meal, it pays to consider the whole menu. (block that metaphor!)

Some Guidelines for applications

1. Keeping data in the groupleSpace is GREAT for debugging an entity, but miserable for compute intensive operations, especially numerical ops, since the groupleSpace is string based. Once the entity works, and the application works, consider using a local cache, if you care at all about speed.

2. Rewriting entities one at a time allowed us to minimize how much of the application was unstable throughout development.

3. Infrequently called primitives can be bundled into any entity (this is why any entity can be a space, because calls to the space primitives are infrequent), but compute intensive primitives (rendering, for a hungry example) are better off as autonomous entities (hopefully running somewhere else.)

4. Network hits should be minimized, and more importantly, NEVER CARE IF A MESSAGE GETS DELIVERED. Don't care. Really. Communication is soooo much easier and faster if you never wait for confirmation or an answer. Sure, it's not a dialogue, but yelling is often an effective data dispersal mechanism.

Bundling it all up -- the shell

The SHELL is the framework for an entity. The shell provides the three basic functions described above, process control, data-space, and communications. The shell maintains a local cache of data found in the groupleSpace at initialization.

This initial data is referred to the configuration file of a shell (a ".fig" file), and is loaded into the groupleSpace upon invocation of the shell. A single entity executable may have several unique invocations, each with a distinct .fig file.

This data is defined to be specific to each entity, because it includes the entity's UID (for Unique IDentifier,) which is also the entity's network address. The shell makes heavy use of the initial configuration data, the cache is an optimization.

The SHELL contains the facility for defining public functions. In this implementation, primitives are made public by defining them in the JUMP TABLE. The jump table maps function names to function pointers. This table is resolved at compile time, which means that entities cannot add primitives at run time.

DIGRESSION: rationale of using a compile time resolved jump table

Though dld (written by wilson ho, ucsc, available from UCSC) provides run time mapping of strings to function pointers, it only runs on sparc and vax, not the dec 5000's. The compile time resolution of the jump table is a compromise of extensibility to portability. A dld shell was used at one phase of development, and *you* could write one if you wanted one.

The veos does not provide any notion of restrictions/protections/authentication. These are application programmers concerns. That is, once a primitive is made public, *any* other entity may call it.

The Shell initializes the data-space, sets up the local caches, handles user interrupts, and calls primitives. Functionally separate tasks, such as communication and data-space maintenance are carried out by external primitives linked in by the loader. It is not strictly necessary to use the shell to use parts of the veos, but this document assumes and recommend that you will. heh, heh, heh.

Entities communicate with each other via communication primitives of the talk library. The data format of communication is the data format of a primitive call is the data format of the grouple-space, is a grouple.