Network Objects
(the Joys of Distributed Objects)

by Alexander H. Emmet

Motivations

- We have object oriented programming languages and want to access them over the network
- We have RPC, but want a simpler interface for accessing objects

Two Very Semantically Different Solutions

- Instead of method/function calls, why not use virtual objects (surrogates) which call RPCs for us?

vs. Linda's

- Create tuples in tuple-space which can act as objects or processes (which turn into objects).

Why Linda?

Concurrency when obtained by:

- **Messages** - Passing is inefficient, single point of failure is bad.
- **Object oriented** - Nothing more than glorified message passing
- **Logic** - Higher level, just as ugly, maybe not as flexible
- **Functional** - You're only as good as Compiler

Hello Linda

What is tuple-space?

Something like a P2P network of linked machines, sharing tuples instead of movies and mp3s. It uses distributed hash mapping to keep track of what is in it.

Vs. S/Net's Linda

Exploring Tuple-Space

Linda does exact pattern matching to determine retrieved tuples using a ("Value", 1, ? x) syntax. This is a simple case and these most likely would be far more complex to specify which process had made them, etc.

The Power of Tuples

- Out – Create a tuple
- In – Eat a tuple (it's data is now yours)
- Rd – Read the tuple's values
- Eval – Execute data until it tuple-izes (think spawn thread, but with a tuple payload)

Uses

Instead of messages, you pass tuples. You can just out tuples to make a queue. Since you must In a passive tuple it acts like it's own lock, or can be used as a ticket.

Example

```
Philosopher(i) int i; {
while(1) {
  think();
  in("room ticket");
  in("chopstick", i);
  in("chopstick", (i+1)%Num);
  eat();
  out ("chopstick", (i+1)%Num);
  out("chopstick", i);
  out ("room ticket");
}
}
```

```
Initialize() {
int i;
for (I = 0;i<Num;i++){
  out ("chopstick", i);
  eval(Philosopher(i));
  if (i!=Num-1) {
    out ("room ticket");
  }
}
}
```

S/Net Linda

Used a multicomputer, with a front-end communication processor and a fast word-parallel bus.

Add those four (out, eval, rd and in) to your favorite language.

A Linda preprocessor/compiler creates its own execution paths for these.

S/Net Linda

Has a Linda interrupt handler (for arriving tuples), and an assembly data sender in the C (for ours)

On out(t) t is copied network wide using broadcast

On in(t) it triggers a delete of all other t's before receiving t (this only fails if someone else deletes all t first)

Both in(s) and read(s) block until something matches s

Objects are stored as tuple blocks with network headers (we can just ship it over the network)

Linda Stirs (Jini)

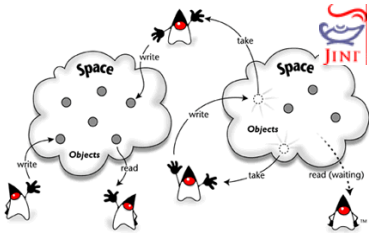


Figure 1. Processes use spaces and simple operations to coordinate activities
Copyright Sun Microsystems, Inc.

Modern Issues?

Comparable to UPnP (security issues).

Can download code from other members.

UDDI and the web services model (but published by a trusted source)

Universal Description, Discovery, and Integration

Why is Jini not breaking out?

Rendezvous (Apple), Web Services, Additionally…

Problems

- Reliability
- Does it scale well for the programmer (# of tuples we might match?
- Greedy/Malicious machines
- Overhead
- Who created the code of my eval tuple?
- Security and Authentication
- Locality
- Non-event driven (Receiver waiting)
- Why can't we forge a 'lock' tuple?

Problems Cont.

- We can solve the high broadcasting overhead and high space redundancy of S/Net, but then we suffer from poor tuple locality, and while P2P may ease this we still have overhead DHT particularly with a low tuple/computer ratio (vs tuple collision, not mentioned).
- Garbage collecting, when do we decide a passive tuple has outlived it's life? Some architectures have implemented timers, but how are these treated by the network?

Network Objects

Welcome to the Titular header.

Hey, maybe it'd be a good idea not to close these connections after every RPC?

Concerns

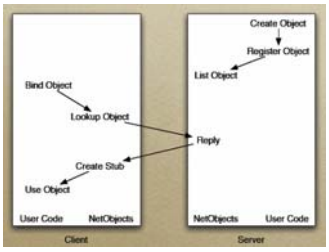
Want to Have

- Type Checking
- Marshaling
- Garbage Collection
- Stream Access

But Ignore

- Transactions
- Object migration
- Distributed shared memory

Interaction Model



Involved in this is the narrowest implementation negotiation.

Stolen from Oliver's slides

Marshaling

Stub mechanisms are hard, but their's uses Pickling.

Theoretically within a factor of two of inline code performance, and adapts to complex structures by using something similar to garbage collection to fully explore them

Need for Types

Typecoded objects (64 bit)

This allows us to 'uniquely' label types to be machine independent.

We don't pass these objects, but surrogate objects, and these when called create pass-through surrogates on the sub-client

Awareness of interprocess streams using a longer duration stub to the server process's stream (Vouchers).

So What's Inside?

Objects include the functions that call them as first parameters

The network object is a 'pure' (method only) object with type T

TSrg – our T's surrogate's type

TImpl – providing the access to the real object

Note that T is not chosen by the programmer, but determined Dynamically, depending on how it is registered on the Owner.

This avoids naming issues on a machine hosting the server and the client.

More Fun

There is transport protocol negotiation when the surrogate is made. (overhead)

Fingerprint (those 64 bits) checks to determine types dynamically.

Vouchers are nonstandard stubs used for streams.

Trash

Garbage Collecting – can't until foreign pointers, are all cleaned (also the client's garbage collector now RPCs the server).

To ensure this the server keeps a set of clients and removes them if they go 'dead' (never reused object IDs, so it can be detected). This prevents memory leaks.

Garbage Collect cycles spanning different address spaces must be broken by the programmer. This can cause memory leaks.

Advantages of Networked Objects

- What does this really buy you?
- Still have RPCs, and they're hidden.
- Binding is handled dynamically.
- Appealing to the OO Programmer

Questions?

Comments?