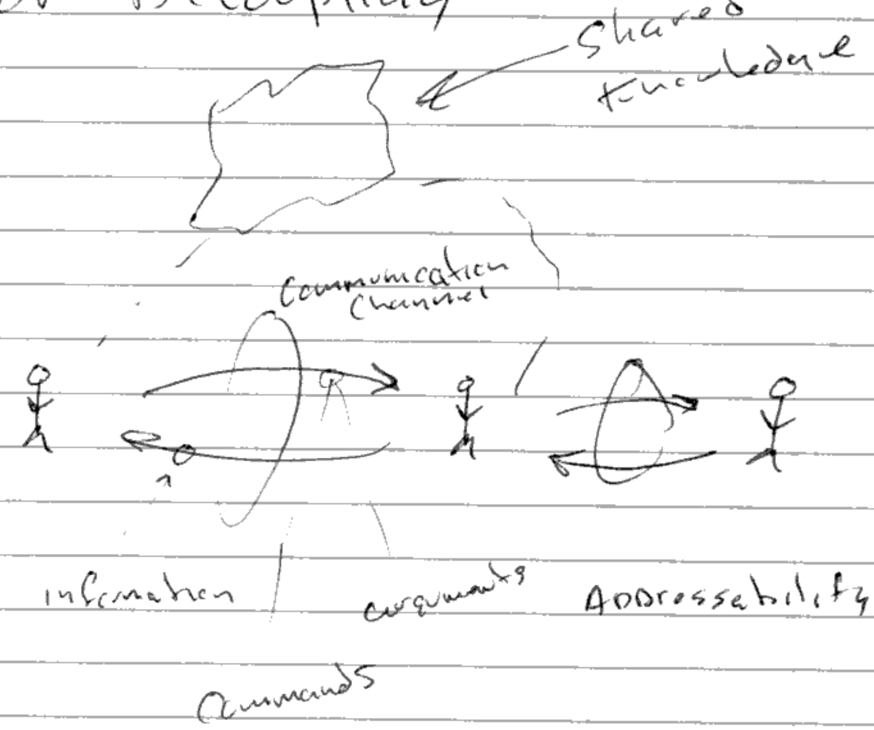
subspace – 001

Actor Decoupling



Transaction

Registration

Actor Decoupling

- Actor placed in completely dark room

Discovery Process

Ping
capabilities in response too
- assumes actors have locations
- assumes locations have connections
- uses this location / space substrate to tunnel to other actors connected to this substrate
- is transactional but different
- location / path to peer actor is carried in transaction

Location substrate is important for all communication

Beacon
- like ping but announcement of location / capabilities not discovery

Registration
- updating location of actor in specific subspace / substrate

The "location" substrate is an enabler for decoupling and multi-instance

Actor Model

Transactions
- require actors to have locations in a substrate
- have state
- require actors to have <u>shared information</u> in terms of responding to messages

What can actors Do

- Bridge between substrates
  - tunnel
  -

- Convert between message encodings

- Project structures from other <u>substrates</u> against each other

Substrate as a concept

-- message is its own substrate

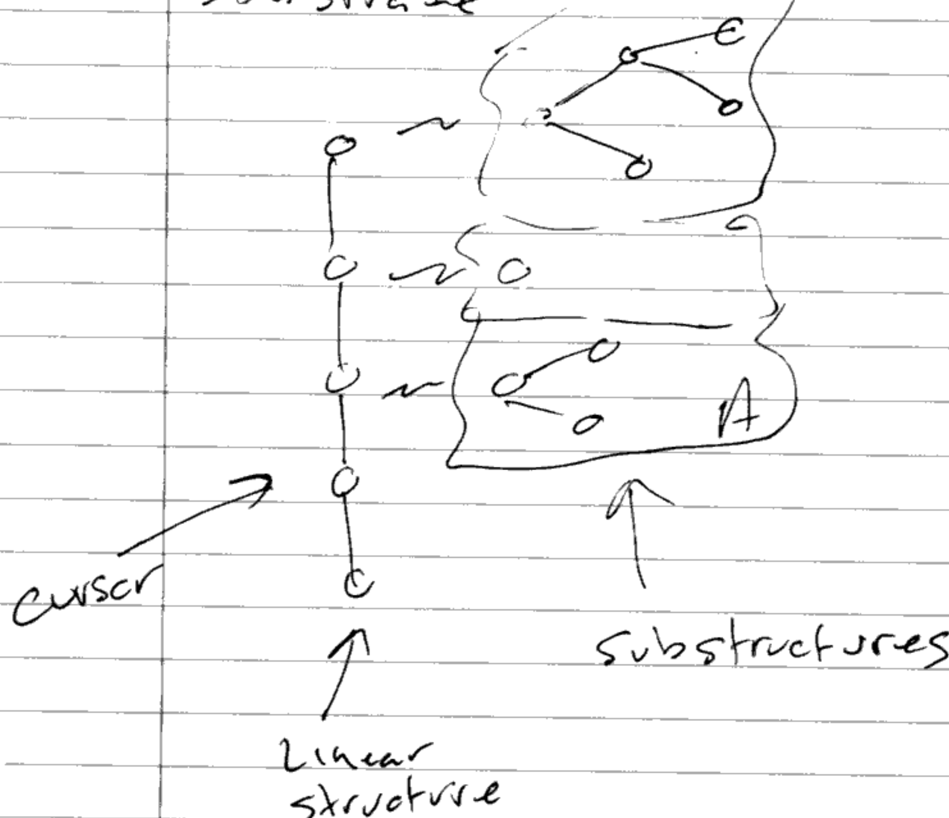
Concept -
Transaction is digging through substrates

Concept
Cursor in substrate

Concept
Function constrains regions and patterns in input/output structures

Concept
Function as a view selector

Substrate



Stacking of substrate

cursor
linear
substrate

Substrate traversal

Cursor opens "access to
Substrate "A"

The message is its own substrate

Thought experiment

Actor / message model assumes messages are snapshots or copies of substrates / subspaces

What if we viewed the message as having access to the subspaces constrained by the message

What if multiple actors could modify the space?

What if copy on write protocols were used?

Concept of substrates

The message is its own substrate...

- the fields in a message are/have locations where data can be found

- this use of locations is equivalent to actors having locations

- the data structure locations can be accessed the same way other actors can be accessed at locations w/messages

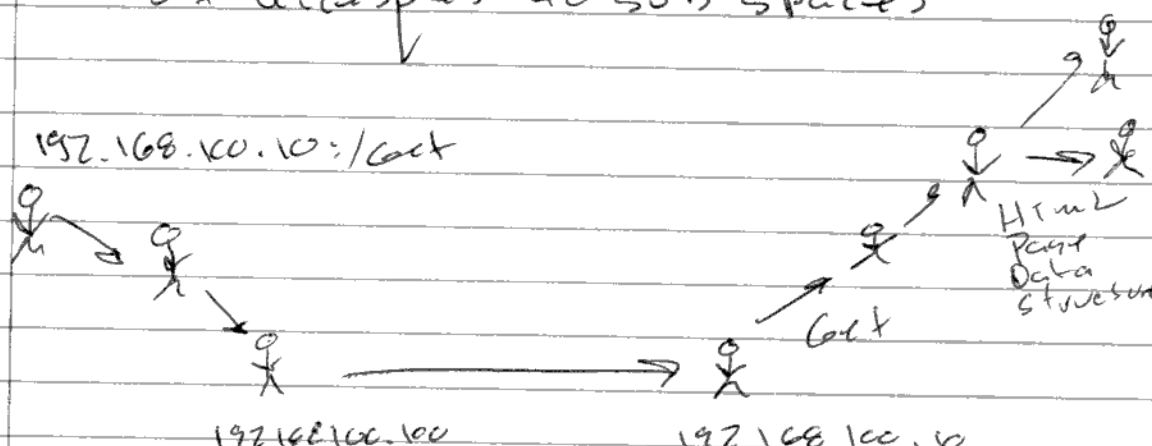
- accessing a location in a data structure just means getting access to the substrate at that location where the substrate may contain another data structure

Substrate operations

Set operation
combine
divide / partition
criteria

this is solid?

Transaction as a hierarchy or sequence of accesses to sub spaces



The method name, "GET" is just another location in a subspace of methods

In this case the GET creates a new, copy generated hierarchical subspace set as a new Data structure by aggregating selected sub spaces

So...

our functions/methods are just the equivalent of location identifiers in some subspace/substratum.

Something that makes functions/metods unique is the requirement to synthesize a new ~~sub~~ hierarchical subspace when the function/metod is called.

Conventional Data structures are more static, and dont require the synthesis of the hierarchical subspace when accessed

This is probably related to copy on write behavior.

Once the hierarchical subspaces/actors in a data structure are created they can be accessed (read) repeatedly w/o being reconstructed

How are functions / metods
different than data structure
access if both are just
steping through a hierarchy
of subspaces to get to a
space of interest? ? ?

$z = f(x, y)$

In this example
x, y and f all constrain /
select the hierarchical subspace
z

x and y each constrain down
a hierarchical subspace

function f create subspace z
by traversing x and y

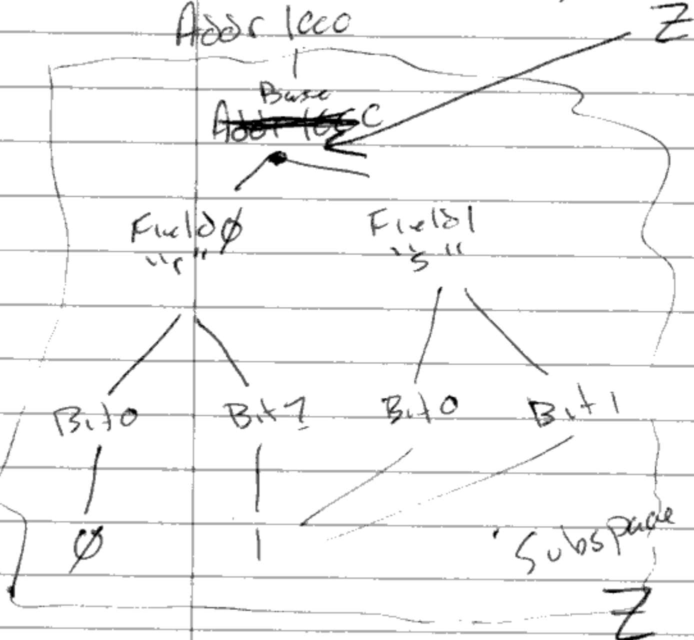
function f only makes sense
if x and y have internal subspaces /
locations known and consistent w
function f

f is modular in that f
works w/ many x's and y's
subspaces

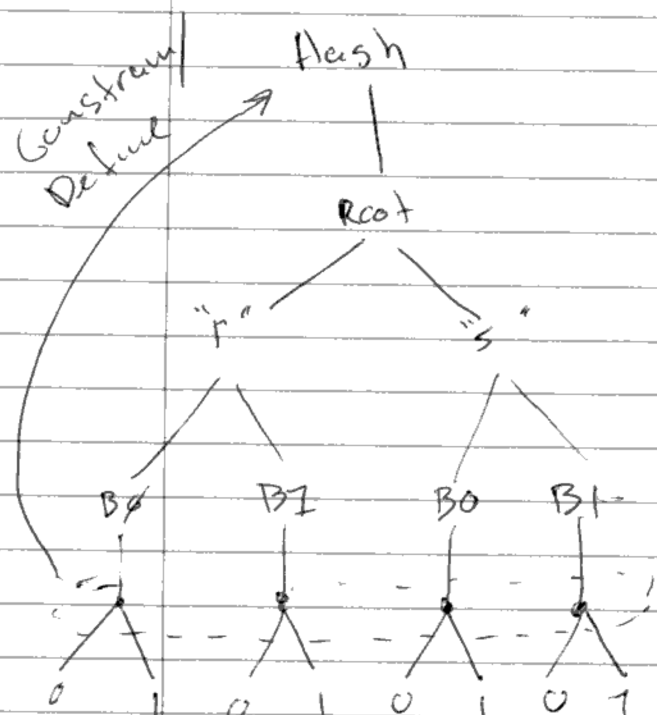
So...

a function/method
- describes/captures/modularizes
- the transform to create subspace z from x, y
  $z = f(x, y)$
- instanciates/~~copies~~ constructs subspace Z
- constrains x and y to be a specific section of hierarchical structure so that f can traverse the structure
- but f does not constrain x and y to a specific instance

Z = f(addr=1000)



Data Structure



A function defines how input spaces are constrained to produce an output space

Function execution is applying the constraints to a region in the hierarchical input structures

The result of function execution on subspaces that are not fully constrained is another function and an input set/space limited to the subspace(s) that are not fully constrained

If/when the inputs are fully constrained the output is a subspace hierarchy that is fully constrained or defined.

Interestingly .....

A fully constrained h subspace
- Can be described as $z = f()$
(no inputs)

But...

A fully constrained h subspace
is actually not observable
and no longer seems to exist

For a subspace to be visible and
interesting to us the h subspace
must be traversable.

Differentiation / Differences
must exist in the h subspace

Minimum traversability looks like

$Z_L = f(\text{position} = L)$ or $Z_R = f(\text{position} = R)$

In other words minimum perseptability
requires

Observable
output
hspace
= Transform Function (at least one input
~~Uncons~~
that is not completely
constrained)

Simplification w/o complete convergence

$b = f(a)$

a = n subspaces

b = n subspaces

$a \rightarrow$ fully ~~converged~~ constrained

then

$b \rightarrow$ imperceptable

$$b = f_b(fa(fa'(fa''(\ldots))))$$

some can be fully constrained

some, at least one must remain unconstrained

$$\Downarrow$$

$$b_\Delta = f_{b\Delta}(f_{constrained}, f_{unconstrained})$$

this new function can be identified/referenced by a hash