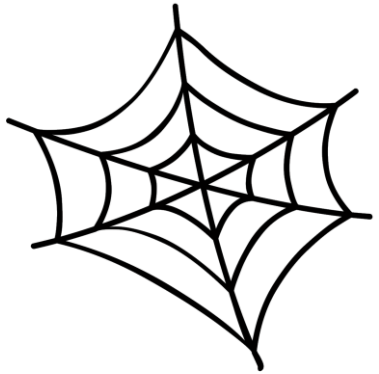
Argonne NATIONAL LABORATORY

BEST PRACTICES (GOTCHAS, TIPS, AND DEBUGGING)



PHILIP CARNS
Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

February 6, 2020

NETWORK FABRIC SUPPORT IN MOCHI

MERCURY NETWORK SUPPORT

"NA" plugins interface with communication libraries

- **Libfabric**
  - Most important library for remote communication
  - Actively supported and improving
  - Supports many networks

Thank you to Intel for contributing the libfabric driver in Mercury.

- **NA SM**
  - Mercury's own transport for shared memory, on-node communication
  - Automatically used when "remote" peers are actually reachable on local node
- **Self RPCs**
  - Not really a transport, but a fast execution mechanism for in-process RPCs
- MPI: strictly for prototyping, not performant
- BMI: legacy code, stable but not recommended for long-term use

See example: mochi-boot-camp/ecp-am-2020/sessions/hands-on/libfabric-config

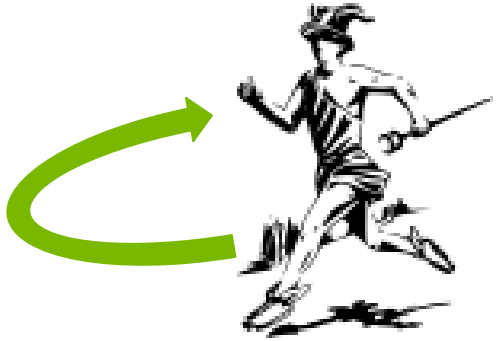
CONFIGURING MOCHI NETWORK TRANSPORTS

Steps to validate configuration for a new network:

- Build libfabric with the correct providers compiled in (more on this in later slides)
  - Add "`variant: fabrics=<list>`" to libfabric section of packages.yaml or add "`fabrics=<list>`" to spack install command line
- Load libfabric package with "`spack load -r libfabric`"
- Use "`fi_info`" command line tool to list available interfaces
  - Intersection of protocols supported by your build and physical interface probe
  - Make sure to run it on the correct node (compute nodes may not use the same fabric as login or management nodes).
  - Make sure that libfabric can access the interface(s) that you expect
- Load Mochi software
- At runtime, use appropriate protocol name (e.g., "`verbs://`") to initialize Margo

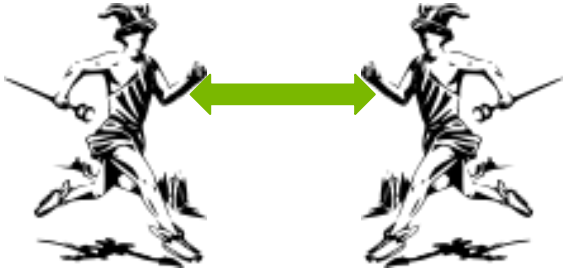
MERCURY “SELF” RPCS

Not really a transport, but good to know about!



- What if a process sends an RPC to itself?
- Mercury detects this and skips all NA plugins
  - RPC handler functions are invoked directly
  - Still delegated to the same Margo thread pool they would have been
  - Bulk transfers become memory copies
- Why is this important?
  - Flexible composition!
  - Components can communicate with other components using RPCs, regardless of whether peer component is in-process, on-node, or remote
- This functionality is enabled by default

MERCURY NA SM PLUGIN



- Can be selected explicitly using the “na+sm://” network address prefix
- Will also be implicitly used when a remote process is detected locally
- On Linux:
  - Uses Cross-Memory Attach capability (single copy)
  - If bulk transfers fail, on some system you may need (as root): “`echo 0 > /proc/sys/kernel/yama/ptrace_scope`”
- Others:
  - Uses conventional POSIX shared memory (2 copies)
- Enabled by default in spack or with NA_USE_SM variable in cmake

MERCURY LIBFABRIC: IB/VERBS

The essentials

- Address prefix: “verbs://”
  - Mercury will internally translate that to a more verbose “ofi+verbs;ofi_rxm://”
- Libfabric providers that will be used: “verbs” and “rxm”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/cooley
- packages.yaml:

```
mercury:
        variants: ~boostsys+ofi
libfabric:
        variants: fabrics=verbs,rxm
```

MERCURY LIBFABRIC: IB/VERBS

Gotchas

- If using libfabric version >= 1.8: "`export FI_MR_CACHE_MAX_COUNT=0`"
  - Disables new memory registration cache, which has some performance flaws that impact Mochi (for now, hopefully fixed soon)
- If using libfabric versions < 1.8: "`export FI_FORK_UNSAFE=1`"
  - Mochi doesn't actually care about fork; this is a compatibility issue with MPI
- Aside from above caveats, works great! Fast and stable.
- Performance tuning: be aware that memory registration can be relatively expensive (margo_bulk_create()), especially for small transfers
  - We don't have a silver bullet yet, but for some size ranges we have better performance by copying through pre-pinned buffer rather than pinning in place

POLLING MODES (ANY TRANSPORT)

Hey network card: Are you done? Are you done? Are you done?

- Ok, how about now?
- Generally speaking, any of the network transports can be driven in “polling mode”, where a CPU core constantly checks for progress.
  - This is fast - no interrupt or context switch when messages complete
  - This also eats a CPU core. The value of this tradeoff depends on the use case
  - Benchmarks almost always busy poll.
- Margo defaults to an adaptive model: it idles gracefully when there is nothing to do, switches to busy mode under load.
- You can force it to always busy poll by setting progress_mode to NA_NO_BLOCK

https://xgitlab.cels.anl.gov/sds/sds-tests/blob/master/perf-regression/margo-p2p-latency.c#L91

THE STATE OF MOCHI TRANSPORTS

February 2020

- Things are continually improving, particularly in the interaction between Mochi, libfabric, and MPI
- The libfabric verbs provider in particular is advancing rapidly, look for big enhancements in the next version after 1.9.0
- There are usually some compatibility quirks.
- Want to keep up over time? Your best option right now is to monitor the scripts that we use for regression testing on different transports:

https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression

- We update these over time with best practice for platforms at ANL

DEBUGGING

DEBUGGING MOCHI COMPONENTS

Debugging Mochi components can be challenging for a few reasons:

- Spack build process
  - Spack intentionally abstracts many of the build steps
  - How do you (for example) add debugging symbols?
- User-level threading
  - Argobots saves and restores contexts in user space
  - This can confuse debugging tools that don't realize when the stack has been replaced, for example: could lead to false positives
- Many components are bleeding edge
  - Understanding how to switch them out to isolate problems can be helpful

CHANGING SPACK BUILD OPTIONS

E.g. adding debugging symbols

- If you want to set specific CFLAGS for a particular stack build (a “concretization” in Spack terminology), you can do it on the command line:

```
[pcarns@carns-x1 ~]$ spack install margo cflags="-g -fno-omit-frame-pointer"
```

- IMPORTANT: if you already had margo installed, this will give you ***another*** margo build, with different cflags on the entire stack. Load either one by specifying the cflags on the load command, or referencing specific hashes
  - Example on next slide

CHANGING SPACK BUILD OPTIONS

E.g. adding debugging symbols

```
[pcarns@carns-x1 ~]$ spack load -r margo
==> Error: the constraint '['margo']' matches multiple packages:
            vomtnss margo@0.5.2%gcc@8.3.0 arch=linux-ubuntu19.04-x86_64
            iwplbrr margo@0.5.2%gcc@8.3.0 cflags="-g -fno-omit-frame-pointer"
arch=linux-ubuntu19.04-x86_64

==> Error: In this context exactly **one** match is needed: please specify your
constraints better.
[pcarns@carns-x1 ~]$ spack load -r margo cflags="-g -fno-omit-frame-pointer"
[pcarns@carns-x1 ~]$ spack unload -r margo cflags="-g -fno-omit-frame-pointer"
[pcarns@carns-x1 ~]$ spack load -r margo/iwplbrr
```

MEMORY DEBUGGING WITH VALGRIND

- As we mentioned before, the problem is that some tools (like gcc's own –fsanitize-address) do not recognize context switches correctly, making their output somewhat unreliable.
- Argobots definitively supports one memory debugging tool, though: **Valgrind**
- Check valgrind documentation if you aren't familiar with it, but the (very) short story is that valgrind reports problems with dynamic memory usage.
  - "valgrind ./my-server –my args"
- To use Valgrind, you must compile Argobots with Valgrind support
- AND make sure that all of your other mochi components use that version of Argobots

MEMORY DEBUGGING WITH VALGRIND

How to enable Valgrind support in Argobots

- Do the following to get a one-off build that depends on Argobots with valgrind support. As in earlier example, this gives you another margo build; you must load the one you want (or uninstall others):

```
[pcarns@carns-x1 ~]$ spack install margo ^argobots@develop+valgrind
```

- Alternatively make a persistent change in your packages.yaml file:

```
argobots:
    variants: +valgrind
```

- This (at least for now) has a performance penalty, even when you aren't using valgrind. This is being improved right now.

DEBUGGING MOCHI COMPONENTS

Isolating component problems

- Many components are bleeding edge
  - A memory corruption in one might show up somewhere else entirely
- Do you suspect a particular component?
  - The suspicion often (rightly or wrongly) falls on networking libraries
  - Try swapping them for debugging purposes to help isolate the source of the problem
- For example: if you compile libfabric with “fabrics=tcp,rxm,sockets” then you have two valid ways to run TCP (from a functionality perspective):
  - Use the “tcp://” prefix on your addresses
  - Use the “sockets://” prefix on your addresses
- Will activate a different code path and help confirm/deny theories

PROCESS AFFINITY

NOT ALL CORES ARE CREATED EQUAL

Computer architectures are increasingly complex, particularly the nodes we typically see in HPC systems:

- Multi-core, multi-socket
- Numa nodes
- Multiple NICs, GPUs
- …

To make most efficient use of these systems, it is important to take note of the locality of these devices and to allocate resources accordingly

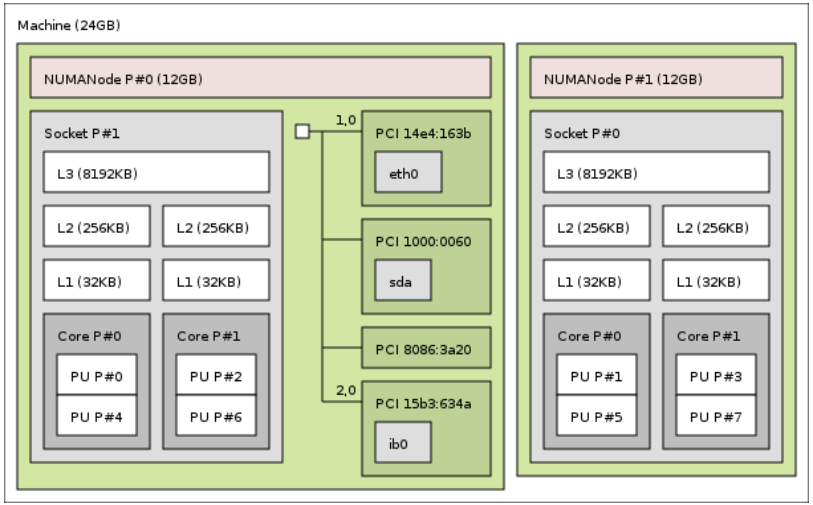
This problem is complicated by multi-threaded applications or by multiple applications/services sharing a node

HOW CAN WE LEARN ABOUT OUR TARGET SYSTEM?

hwloc

- Provides a portable abstraction of the hierarchical topology of modern computer architectures

Using hwloc's 'lstopo' command, we can generate graphical representations of the node architecture (warning, the output may be complex on some systems):



https://xgitlab.cels.anl.gov/sds/mochi-boot-camp

CONTROLLING PROCESSOR AFFINITY FOR MOCHI APPS

Now that you've learned more about your architecture, how can you actually take advantage of it?

Numactl

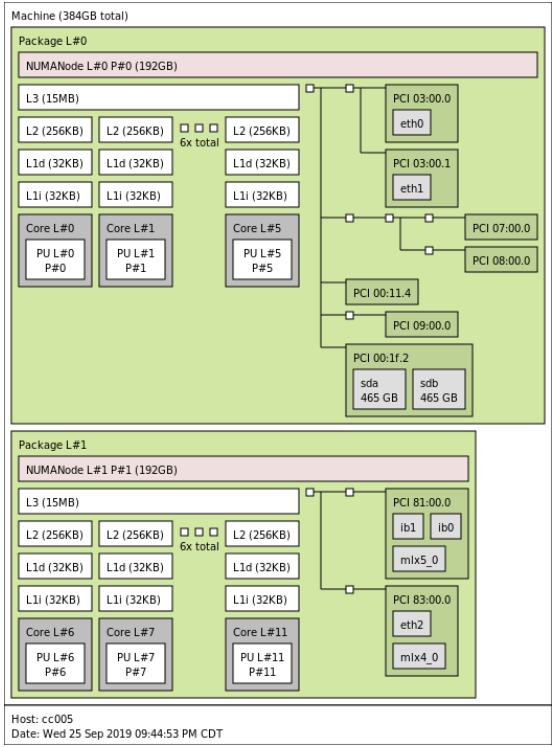
- Allows control of the NUMA scheduling and memory policies for a given executable
- Can specify which cores/sockets to run processes on, as well as which NUMA memory domain these processes allocate memory from
  - --cpunodebind=sockets, -N sockets : only execute on sockets
  - --physcpubind=cpus, -C cpus : only execute on cpus
  - --membind=sockets, -m sockets : only allocate memory from sockets

HWLOC/NUMACTL EXAMPLE

Cooley Linux cluster @ ALCF, which uses an IB network

We prefer to pin a Mochi service on socket 1, since this socket shares locality with the IB controller for this node

- numactl -N 1 -m 1 <executable>



https://xgitlab.cels.anl.gov/sds/mochi-boot-camp

SUPPLEMENTAL MATERIAL:
OTHER NETWORK TRANSPORTS

MERCURY LIBFABRIC: OMNIPATH

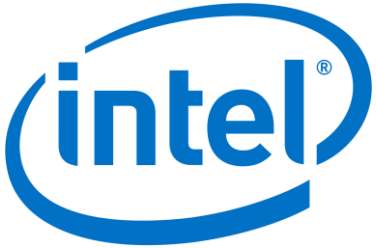
The essentials

- Address prefix: “psm2://”
  - Mercury will internally translate that to a more verbose “ofi+psm2://”
- Libfabric provider that will be used: “psm2”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/bebop
- packages.yaml:

```
mercury:
        variants: ~boostsys+ofi
libfabric:
        variants: fabrics=psm2
```

MERCURY LIBFABRIC: OMNIPATH

Gotchas



- For any version: "`export PSM2_MULTI_EP=1`"
  - Allows Mochi and MPI to share interface if needed
- Performance: excellent
- Stability: YMMV depending on system, library versions, and workload. We don't have universal solution to recommend yet.
- All OmniPath systems also support verbs through emulation, at a performance penalty. In the worst case, use verbs if you have to.
- Memory registration is free (noop), but not really because you pay performance cost at communication time on cold memory access.
- OmniPath/PSM2 consumes more host CPU than other high performance networks.

MERCURY LIBFABRIC: ARIES/GNI

The essentials

- Address prefix: “gni://”
  - Mercury will internally translate that to a more verbose “ofi+gni://”
- Libfabric provider that will be used: “gni”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/theta
- packages.yaml:

```
mercury:
      variants: ~boostsys+ofi
libfabric:
      variants: fabrics=gni
```

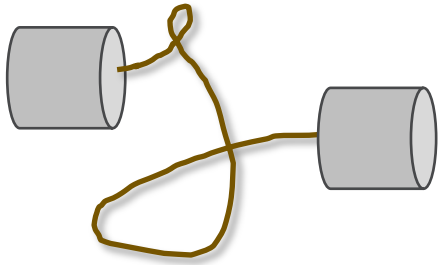
NOTE: This can only be built on a Cray machine, and you might need to load “ugni” module in your environment. Ugni is not a spack package.

MERCURY LIBFABRIC: ARIES/GNI

- "`export MPICH_GNI_NDREG_ENTRIES=1024`"
  - Tells Cray-MPICH to be less aggressive with resource consumption
- Communication across separately launched executables requires either DRC (any launcher) or protection domains (if your system uses aprun)
  - DRC only works with libfabric 1.8.1 or newer and requires some explicit set up in your code (external to Mercury)
- Stability is great
- Performance is (mostly) great but there are gotchas:
  - Latency is poor when not busy-spinning (more on how to do this later)
  - Latency is poor (any polling mode) on KNL cores
  - Memory registration / copy tradeoffs are unclear, possibly similar to IB/Verbs

MERCURY LIBFABRIC: TCP/IP

The essentials

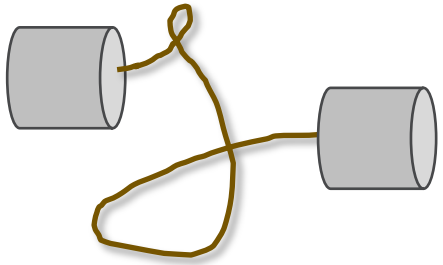


- Address prefix: “tcp://”
  - Mercury will internally translate that to a more verbose “ofi+tcp;ofi_rxm://”
- Libfabric providers that will be used: “tcp” and “rxm”
- Example:
  - https://xgitlab.cels.anl.gov/sds/sds-tests/tree/master/perf-regression/cooley/*tcp*
-

```
mercury:
        variants: ~boostsys+ofi
libfabric:
        variants: fabrics=tcp,rxm
```

NOTE: there are alternative TCP implementations, but this one is our recommendation.

MERCURY LIBFABRIC: TCP



- Please use libfabric version >= 1.9.0
- Stability is good
- Performance is poor (when other alternatives are possible)
  - This is largely because of emulation of RDMA over sockets
- There are other options that you may see in previous examples:
  - “sockets://” activates the legacy/reference sockets implementation in libfabric
  - “bmi+tcp://” activates TCP support in the BMI library (if enabled)
  - All have similar performance characteristics; “tcp;ofi_rxm” is now the most actively maintained, though

SUPPLEMENTAL MATERIAL:
RPC TUNING

BUILT-IN RPC PROFILING

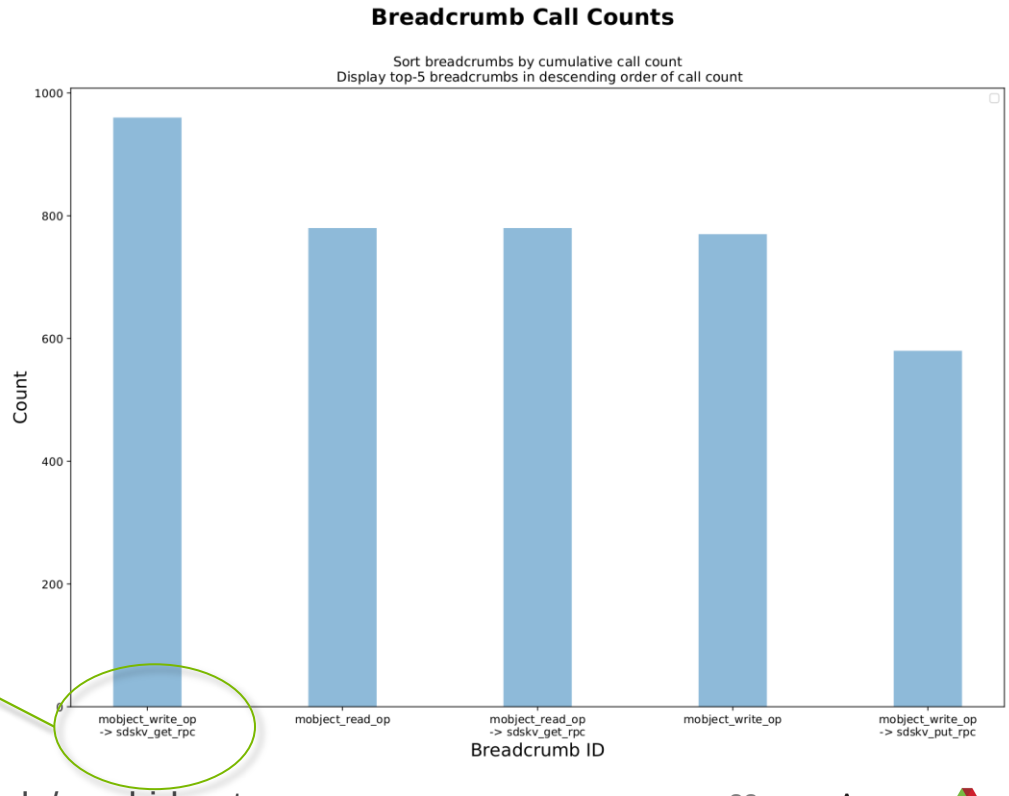
Coming soon!

- How do you tune a Mochi service, or at least understand it's performance?
- Almost every major operation in Mochi is an RPC
  - RPCs can be chained, local, remote, etc.
- If all RPCs in a service were transparently instrumented, then any Mochi service could be quickly characterized
- This is an upcoming feature, thanks in large part to work by Srinivasan Ramesh of the University of Oregon
- Basic usage is modeled after Darshan:
  - Characterization capability is compiled in, no code modification needed
  - Simple command line tools to produce a PDF with plots to use as a starting point for performance tuning

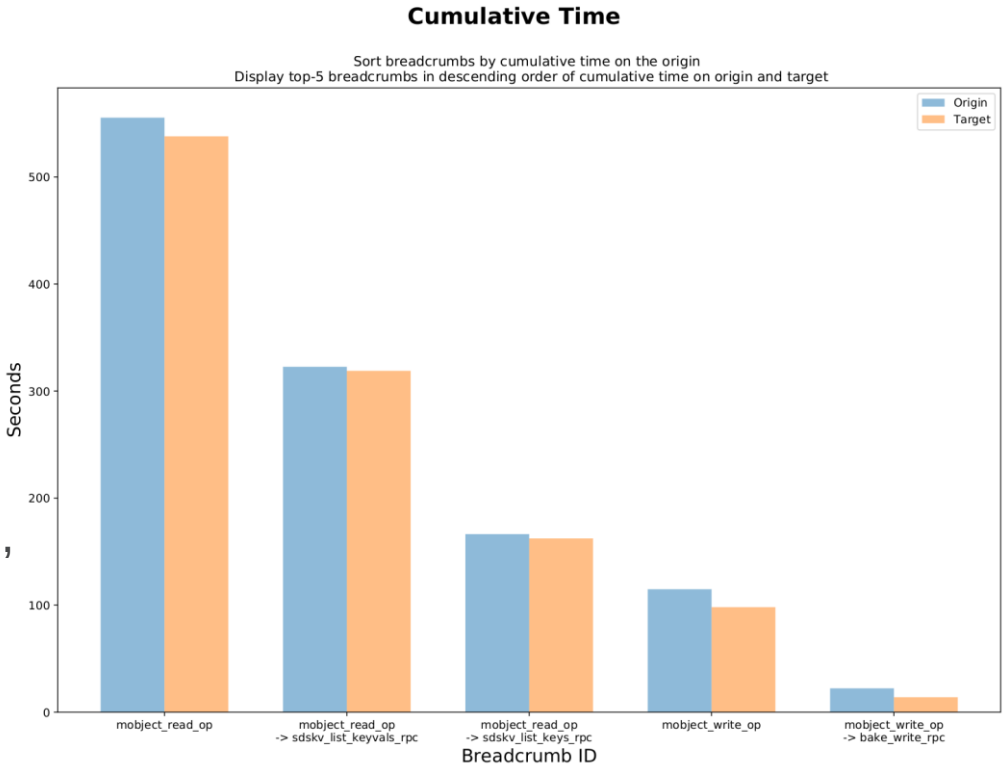
MOCHI RPC BREADCRUMBS

What RPCs are most prolific?

- This graph shows ranked list of 5 most frequently called RPCs
- Each includes lineage if applicable (the chain of RPCs that led to the one being measured)



- This graph shows ranked list of 5 RPCs that cumulatively consumed the most service time
- Split by client and server side elapsed time
- Each includes lineage if applicable, just as in the previous graph.



RPC BREADCRUMBS

Status

- Mochi already includes the mechanism for measuring RPC time and tracking the lineage of RPCs
  - The examples you run today include this capability already
- Remaining work is productizing the tools that can dump summary data and plot it
- There are many other possibilities for deeper analysis from that starting point
- TODO: update this slide, tools are in official releases now too, show command line

THANK YOU!