7TH JLESC WORKSHOP

FAULT DETECTION AND GROUP MEMBERSHIP IN HPC DATA SERVICES

SHANE SNYDER
Argonne National Laboratory
ssnyder@mcs.anl.gov

July 19, 2017
NCSA, Urbana, IL

STATE OF THE ART IN HPC DATA MANAGEMENT

- Apps generally use high-level I/O libraries to transform workloads into a form suitable for storing on a PFS
  - Much work invested to optimize I/O performance for specific workloads/architectures
- However, today's increasingly data-intensive apps are ill-served by PFS designs that have remained mostly stagnant for decades
  - POSIX-compliance
  - Inflexible designs that cannot efficiently leverage hierarchical storage models and emerging storage technologies (e.g., non-volatile)
  - Poor fault detection/recovery

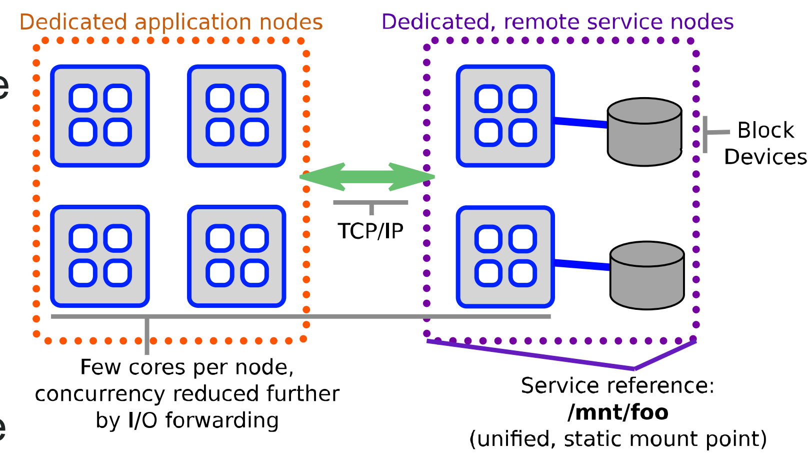


Figure courtesy of P. Carns

Argonne NATIONAL LABORATORY

A PROGRESSIVE APPROACH TO HPC DATA MANAGEMENT

- Instead of a 'one-size-fits-all' PFS, provide specialized services tailored for applications' specific data management requirements
  - Approach based on composable, re-usable, and lightweight **microservices** for data management
    - Services could be generic, supporting classical HPC data management techniques (checkpoint-restart, in-situ)
    - Or, app-specific services to handle workloads that have traditionally been problematic

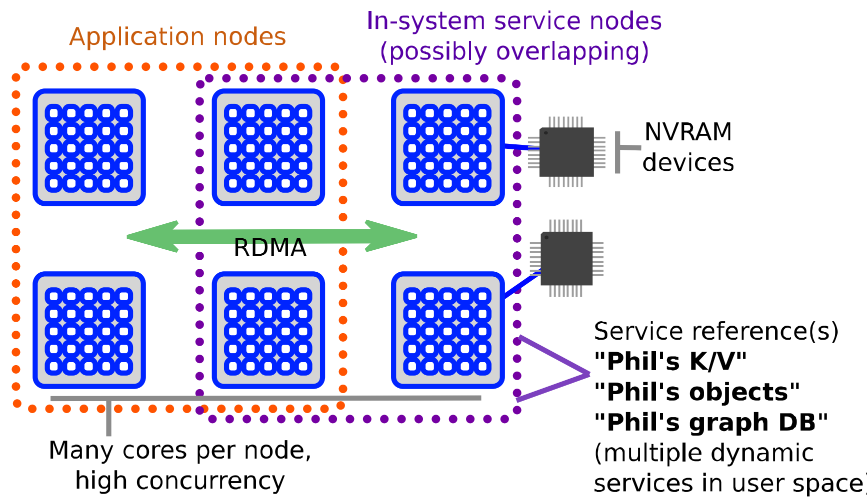


Figure courtesy of P. Carns

THE MOCHI PROJECT

- *Vision*: Enable rapid development/deployment of efficient HPC data management services based on the microservice model we have described
  - Develop re-usable building block microservices that form the foundation of many distinct HPC data services
    - Key-val stores, distributed object stores, pub-sub systems
  - Adapt generic microservices to satisfy application needs (e.g., scale, consistency model) and to efficiently utilize available system resources (storage hierarchies, burst buffers)
  - Present a coherent data management API that supports an application's native data model while abstracting low-level service details

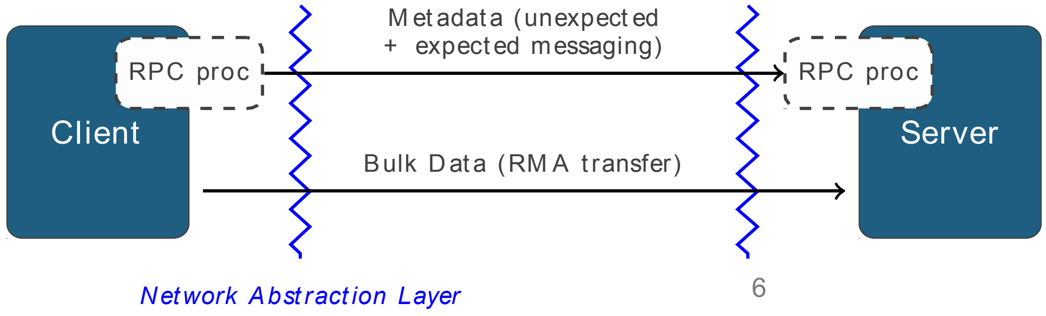
The Mochi project is a collaboration between ANL, The HDF Group, LANL, & CMU
http://www.mcs.anl.gov/research/projects/mochi/

MOCHI: ENABLING TECHNOLOGY

COMMUNICATION: MERCURY

A high-performance RPC framework for HPC systems

- Mercury is an RPC system for use in the development of high performance system services.
  - Portable across systems and network technologies
  - Efficient bulk data movement to complement control messages
  - Provides simplifications for service implementers:
    - Remote procedure calls
    - RDMA abstraction (or emulation)
    - Protocol encoding
    - Clearly defined progress/event model
    - No client/server role restrictions
    - No global fault domain (MPI_COMM_WORLD)

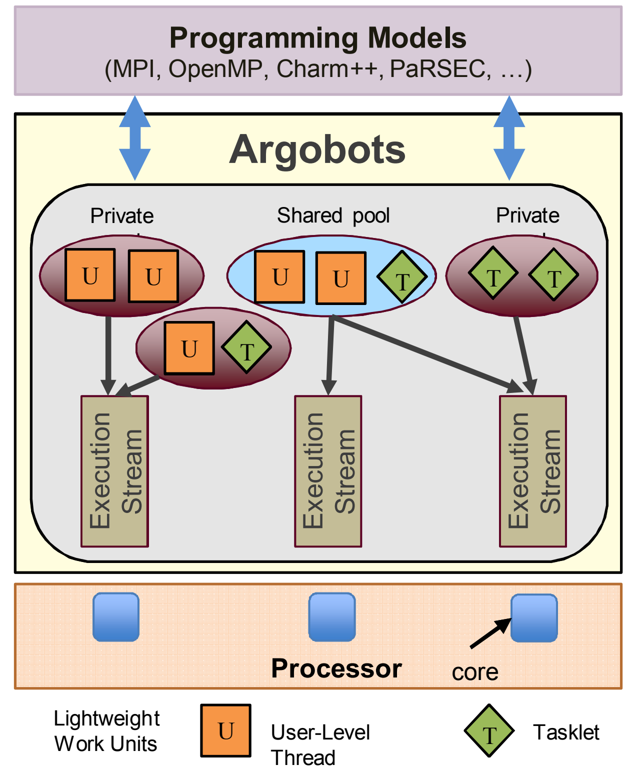


Developed by The HDF Group & ANL.
https://mercury-hpc.github.io/

CONCURRENCY: ARGOBOTS

A lightweight threading/tasking framework

- User-level threading: lightweight context switching among many concurrent threads
- Use multiple cores and control delegation of work to those cores
- Key features for data services:
  - Lets us track state of **many** concurrent operations with simple service code paths and low OS resource consumption
  - Custom schedulers (i.e., to implement priorities, or limit CPU usage)
  - Primitives that facilitate linkage to external resources



http://argobots.org/

FAULT DETECTION AND GROUP MEMBERSHIP AS A MICROSERVICE

SSG

Scalable Service Groups

- A generic group membership service developed as part of the Mochi project
- Allows for organizing sets of Mercury endpoints into logical, fault-tolerant process groups
  - These groups provide basis for deploying and referencing distributed services
- Key functionality:
  - Bootstrapping process groups
    - Available now: MPI communicator and config file based bootstrapping routines
    - Future work: PMIx bootstrapping for production systems
  - Maintaining collective group membership state as members dynamically join/leave
    - At minimum, state includes the membership **view**, a mapping of group member IDs → Mercury address information
  - Detecting and notifying users of group member faults

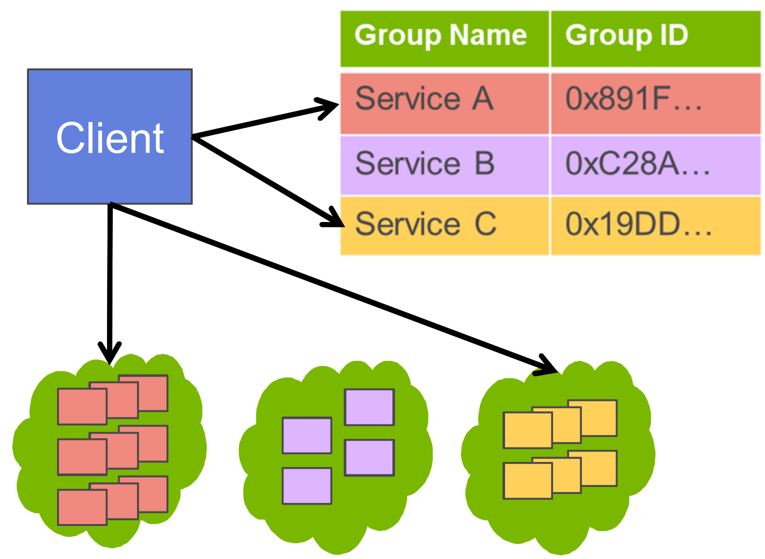
https://xgitlab.cels.anl.gov/sds/ssg

SSG API

- Group bootstrapped by collectively calling `ssg_group_create` at all members
  - Returns opaque ID that uniquely references group, and that internally encodes address info on at least one group member
- Processes can dynamically add/remove themselves from the group using `ssg_group_join` & `ssg_group_leave`
- Non-group member processes can access the group view using `ssg_group_attach`
  - Attachment is envisioned for group "clients" -- i.e., processes that do not want to become proper group members
- Simple group accessor routines like `ssg_get_group_self_id` and `ssg_get_group_size` provide details on a caller's local group view, while `ssg_get_addr` can be called to map a group member's ID to its Mercury address

EXAMPLE SERVICES DEPENDENT ON SSG

- Group communication abstractions
  - Build overlay networks over SSG groups for efficient collective RPCs
  - Useful for common data reduction or broadcast operations
- Pub-sub
  - SSG groups reference publisher groups that subscribers can attach to
- Service registry
  - Apps query a known server address to get a list of services (i.e., SSG groups) currently available (think DNS)



A Mochi service registry indicating location of SSG service groups

SSG FAULT DETECTION & GROUP UPDATES

- SSG includes an implementation of SWIM, a protocol for detecting faults and managing group membership
  - Faults detected by periodically probing random group members for liveness, rather than heartbeats
  - Protocol includes a suspicion mechanism to avoid marking unresponsive members as dead until some timeout has elapsed
  - Membership updates disseminated using a gossip protocol piggybacking on the protocol's ping messages

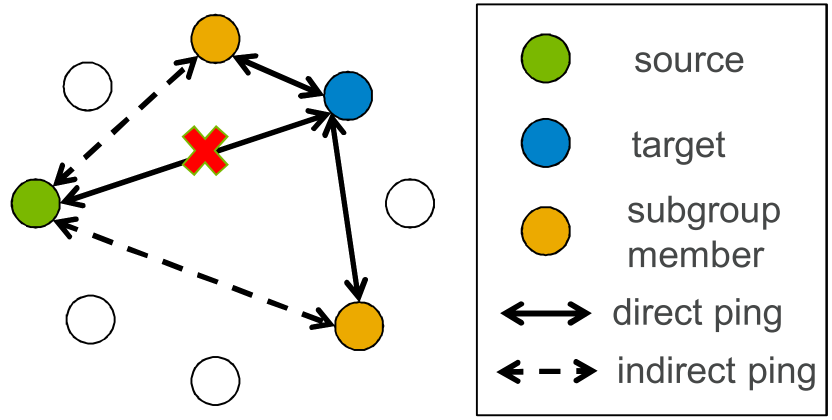
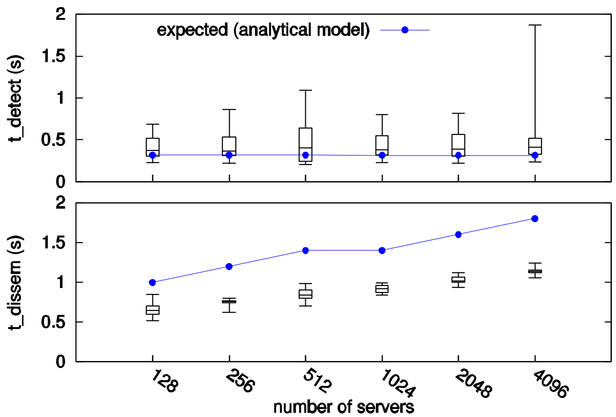


Illustration of SWIM-style failure detection protocol

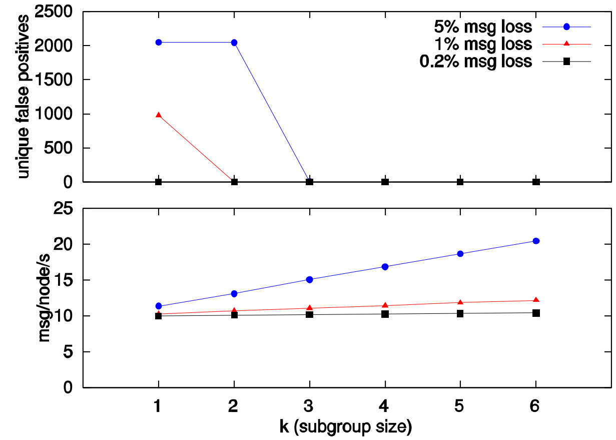
A. Das et al. "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol." DSN ’02. 2002.

SWIM PROTOCOL EVALUATION

- Observations:
  - Failure detection times are mostly constant, irrespective of group size
  - Update dissemination completes in O(log(n)) time, with n = group size
  - Network load scales linearly with subgroup size, accuracy scales exponentially
  - Accuracy can be preserved even in cases of extreme message loss



SWIM protocol failure detection & dissemination time



SWIM protocol false positive rate & network load

Snyder et al. "A Case for Epidemic Fault Detection and Group Membership in HPC Storage Systems." PMBS '14. 2014.

AUGMENTING SWIM

- SWIM uses gossip internally for collectively maintaining the group view -- but could we expose it for maintaining any generic group state?
  - E.g., in a pub-sub service, a publisher group needs to maintain a consistent view of subscribers to push topic updates to
  - Increases communication efficiency, as app-specific group state can just be piggybacked on SWIM's internal messages
- What if an application needs a strongly-consistent view of group membership (i.e., ordering of membership updates is important)?
  - Designate subset of group members to run a RAFT cluster to reach consensus on ordering of group state changes
  - The RAFT cluster can then lazily propagate state changes out to regular group members using SWIM's gossip protocol

D. Ongaro et al. "In Search of an Understandable Consensus Algorithm." USENIX Annual Technical Conference. 2014.

THANK YOU!

THIS WORK WAS SUPPORTED BY THE U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE, ADVANCED SCIENTIFIC COMPUTING RESEARCH, UNDER CONTRACT DE-AC02-06CH11357.