The ABCs of Open MPI

Decoding the Alphabet Soup of the Modern HPC Ecosystem (Part 2)

Ralph H. Castain, Jeffrey M. Squyres

Presented in conjunction
with the EasyBuild community

Webex Logistics

- This session is being recorded
- Ask questions in the Q&A panel

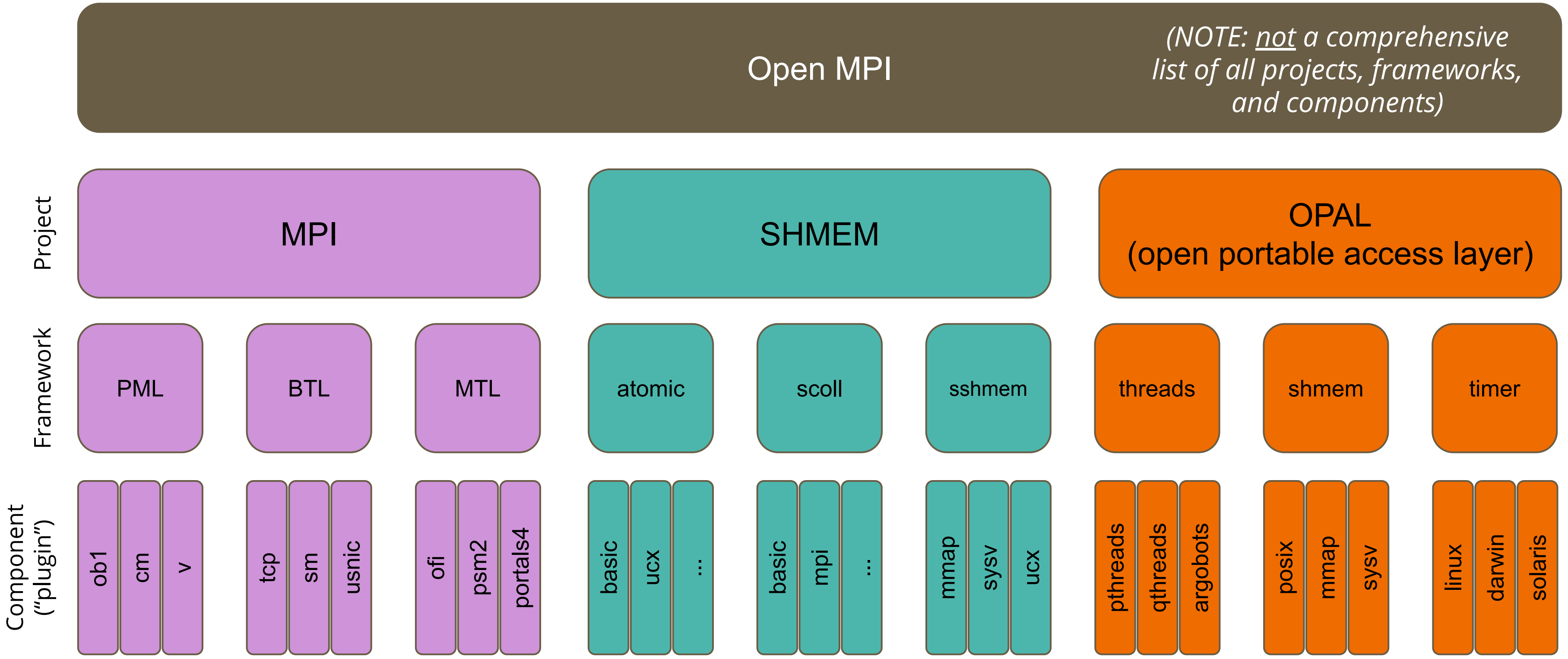
Overview

- Background
- PMIx: What is it?
- Building Open MPI

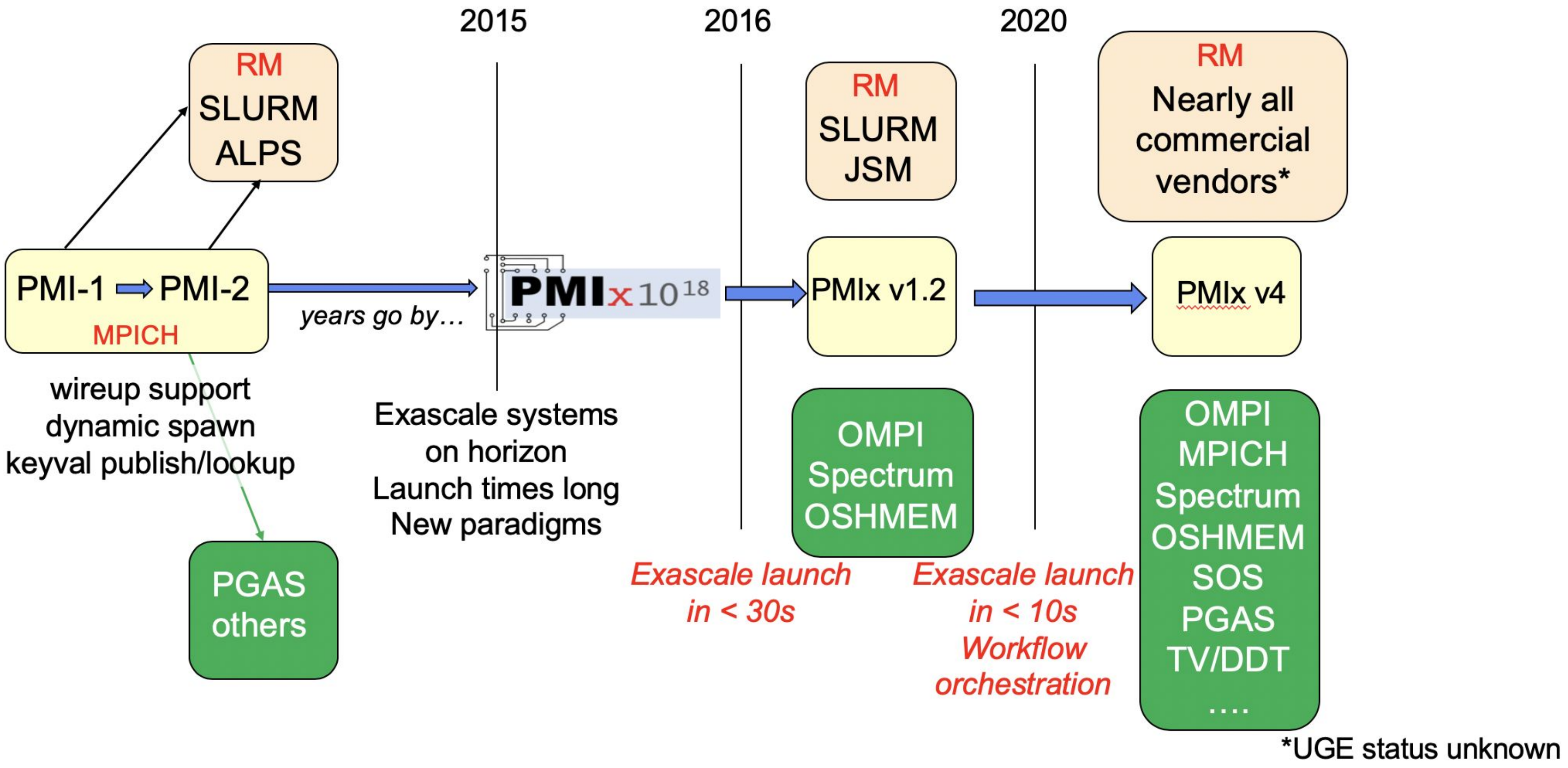
$\Bigg\}$ Covered in part 1

- A breakdown of Open MPI:
  - The run-time stuff
  - The MPI stuff
- Configuration / debugging tips
- The upcoming Open MPI v4.1.x series
- The upcoming Open MPI v5.0.x series

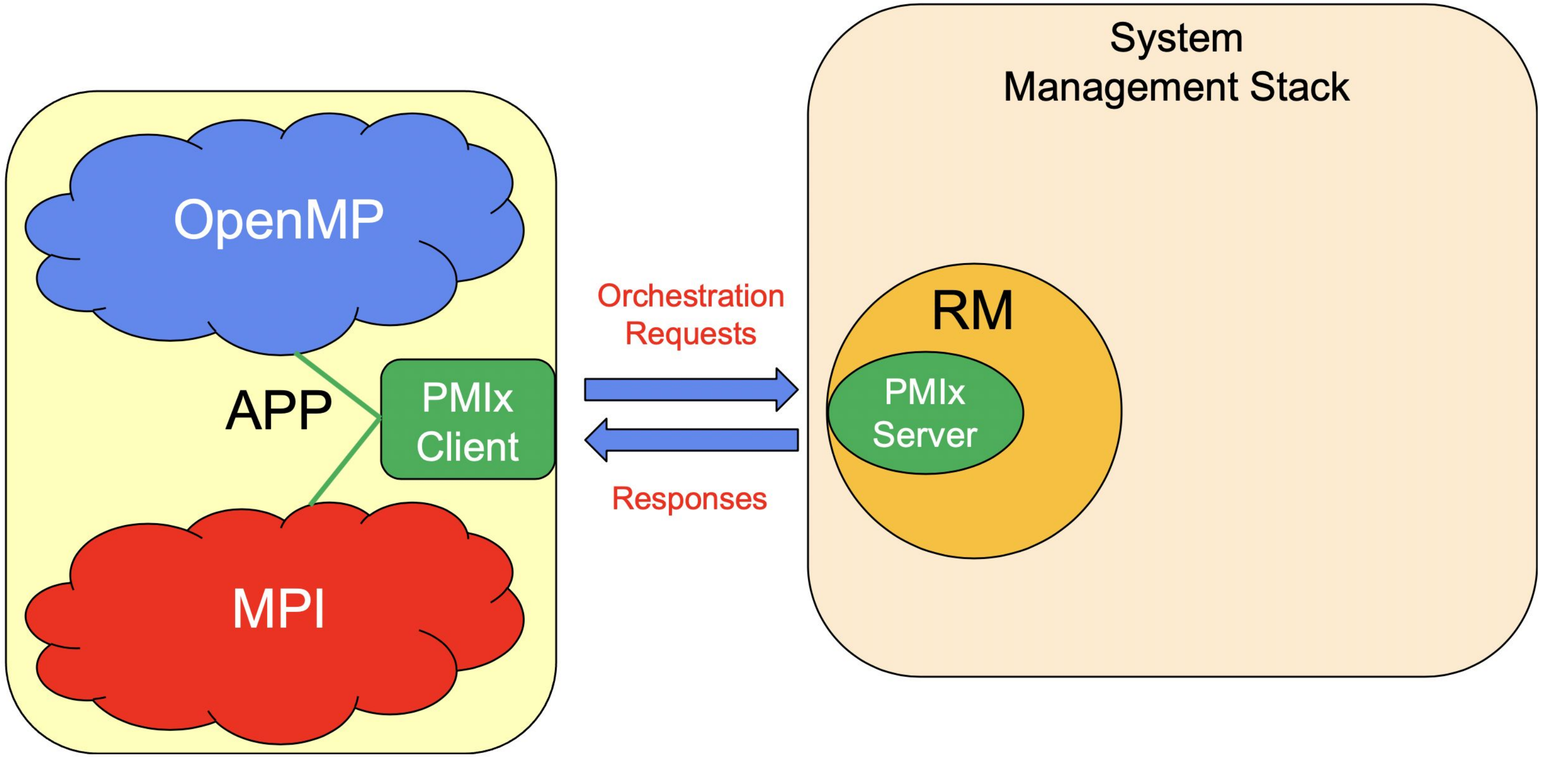
Recap of Part 1: Projects, Frameworks, Components



Recap of Part 1: What is PMIx?



What Is Its Role?



Recap of Part 1: Building Open MPI

```
wget \
  https://download.open-mpi.org/release/open-mpi/vx.y/openmpi-x.y.z.tar.bz2
tar xf openmpi-x.y.z.tar.bz2
cd openmpi-x.y.z

./configure --prefix=$HOME/my-ompi <options> |& tee config.out
# Most <options> typically deal with network communications
# libraries (e.g., libfabric, UCX)

make -j 8      |& tee make.out
make install |& tee install.out
```

Open MPI: The Run-Time Stuff

PMIx/PRRTE & OMPI…Oh My!

Ralph H. Castain

Where Is It Used?

- Libraries
  - OMPI, MPICH, Intel MPI, HPE-MPI, Spectrum MPI, Fujitsu MPI
  - OSHMEM, SOS, OpenSHMEM, …
  - PGAS
- RMs
  - Slurm, Fujitsu, IBM's JSM, PALS (2020), PBSPro (2019), LLNL Flux, Kubernetes (2020)
  - Slurm enhancement (LANL/ECP)
- New use-cases
  - Spark, TensorFlow
  - Debuggers (TotalView, DDT)
  - MPI
    - Re-ordering for load balance (UTK/ECP)
    - Fault management (UTK)
    - On-the-fly session formation/teardown (MPIF)
  - Logging information
  - Containers
    - Singularity, Docker, Amazon

Build Upon It!

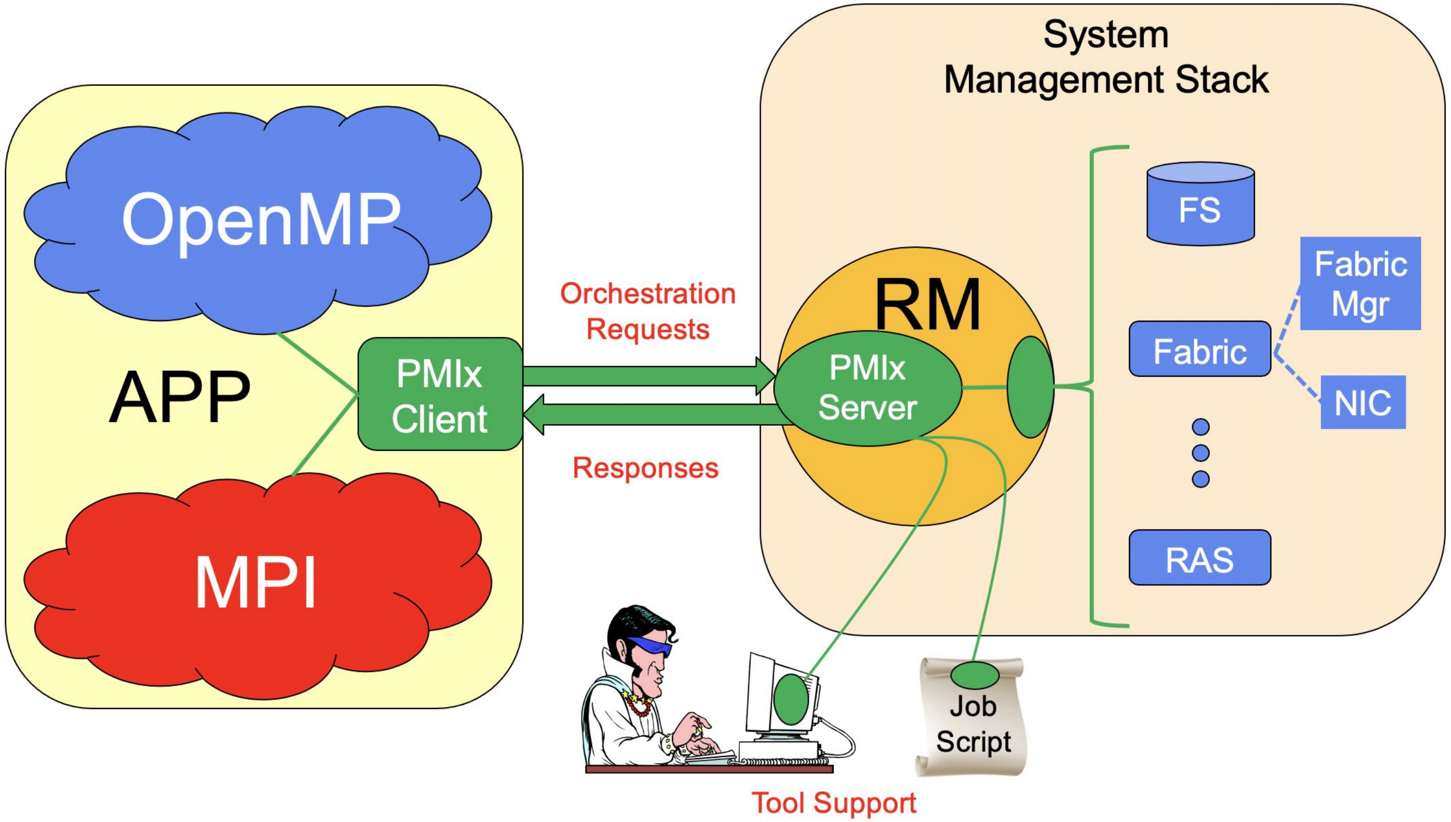
- Async event notification
- Cross-model notification
  - Announce model type, characteristics
  - Coordinate resource utilization, programming blocks
- Generalized tool support
  - Co-launch daemons with job
  - Forward stdio channels
  - Query job, system info, network traffic, process counters, etc.
  - Standardized attachment, launch methods

Sprinkle Some Magic Dust!

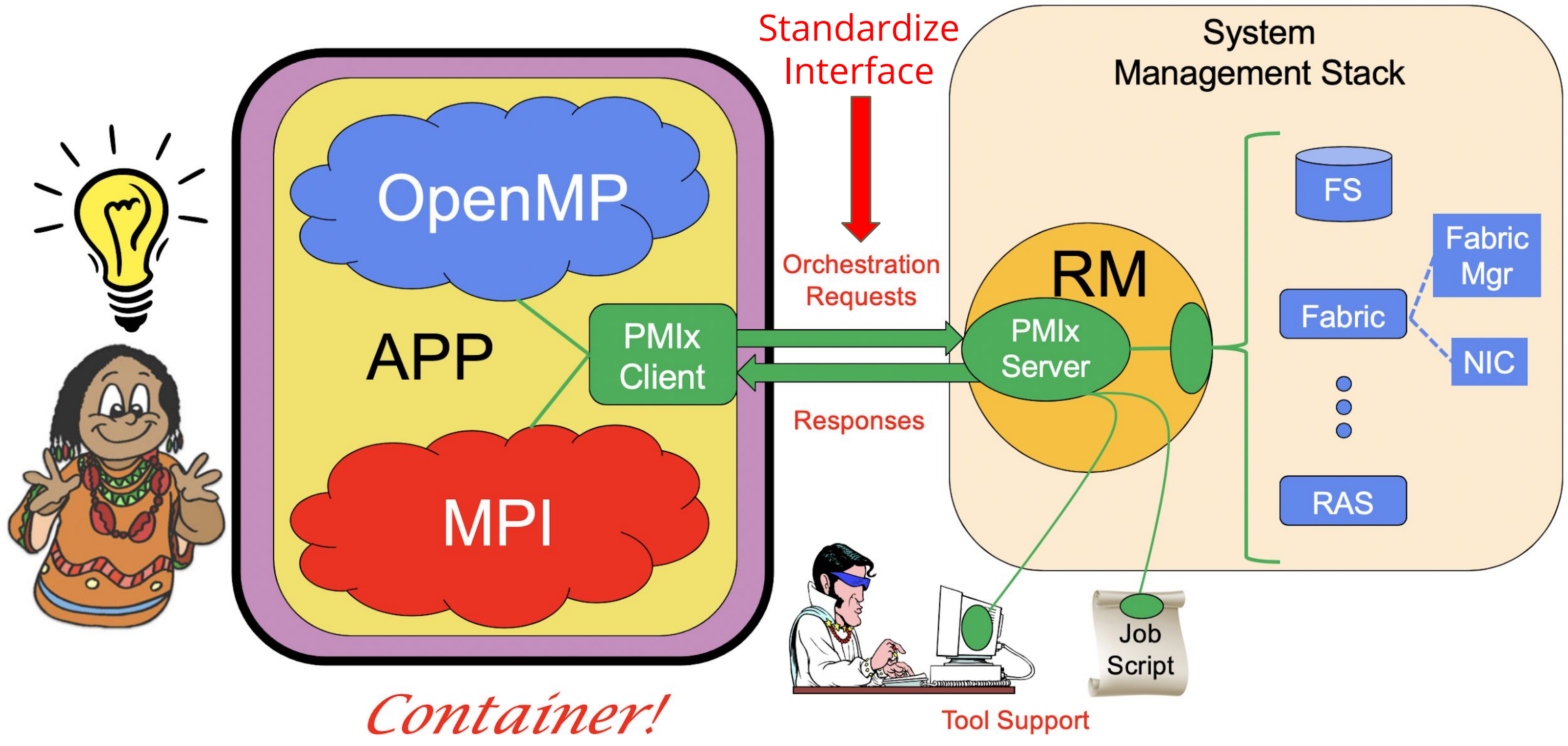
Legion

- Allocation support
  - Dynamically add/remove/loan nodes
  - Register pre-emption acceptance, handshake
- Dynamic process groups
  - Async group construct/destruct
  - Notification of process departure/failure
- Storage integration
  - Pre-cache files, specify storage strategies
- Power management (strategy)

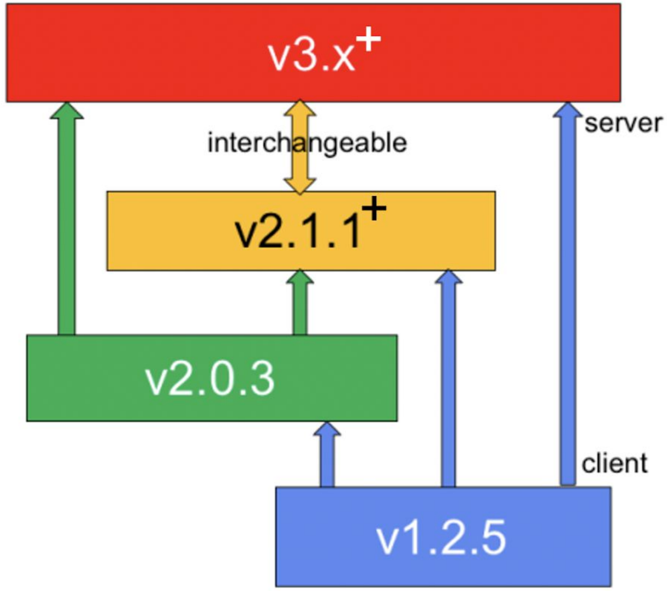
PMIx Integration Architecture



PMIx Integration Architecture

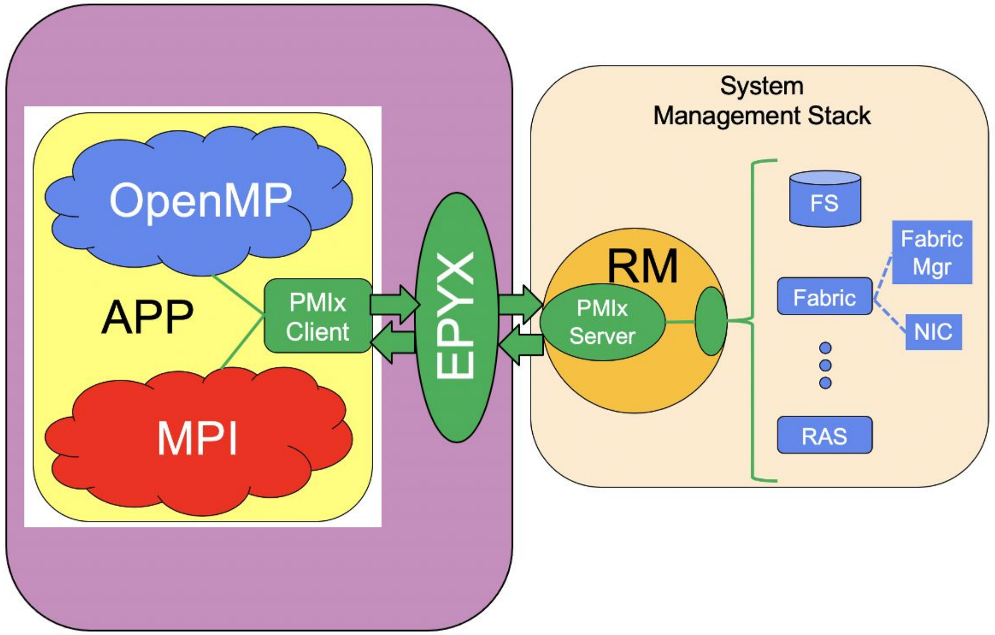


Cross-Version Compatibility



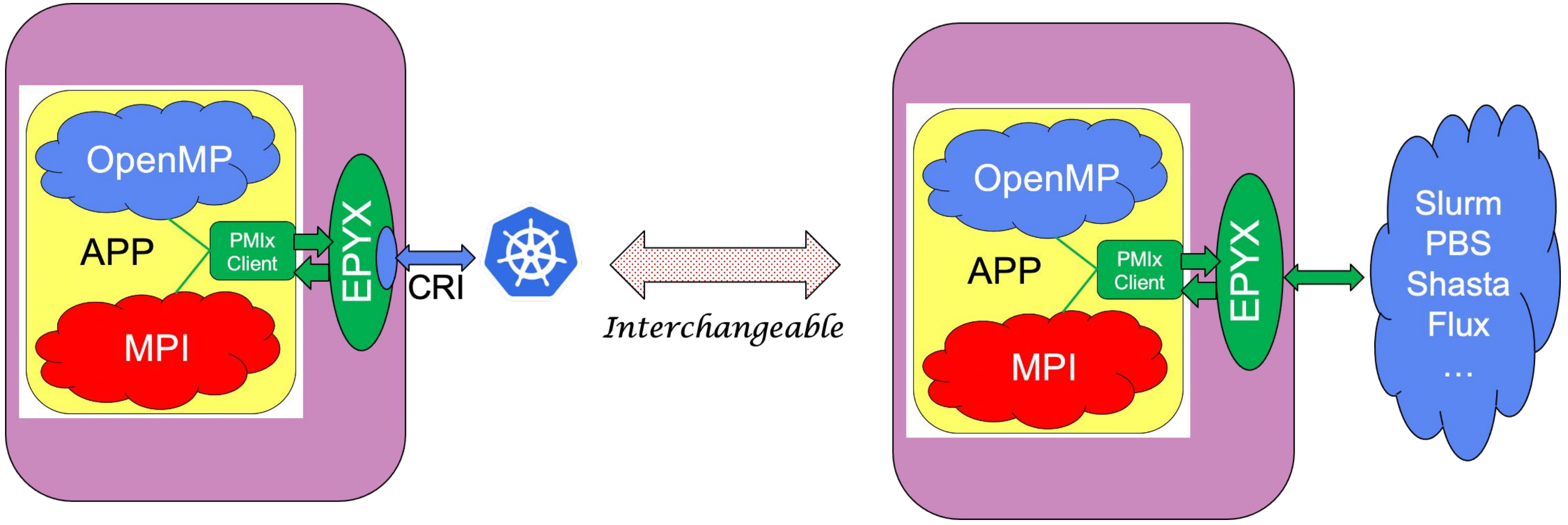
- Auto-negotiate messaging protocol
- Client starts
  - Envar indicates server capabilities
  - Select highest support in common
  - Convey selection in connection handshake
- Server follows client's lead
  - Per-client messaging protocol
  - Support mix of client versions

EPYX: Filling the Gaps

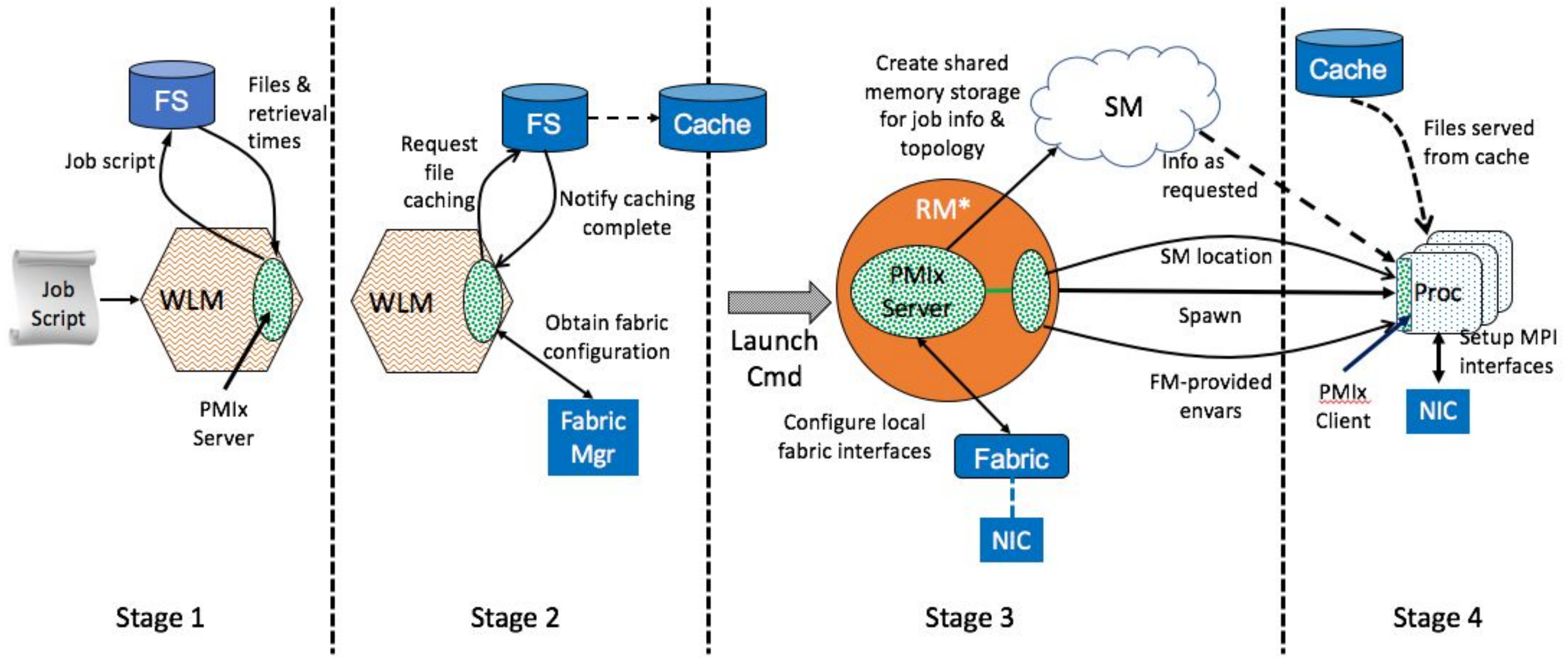


- PMIx relay daemon/server
- Integrated into container
- Sense what SMS supports
  - From nothing to everything
- Supported requests
  - Relay requests/responses
- Unsupported requests
  - Execute internally
  - Return “not supported”

Enhance Portability

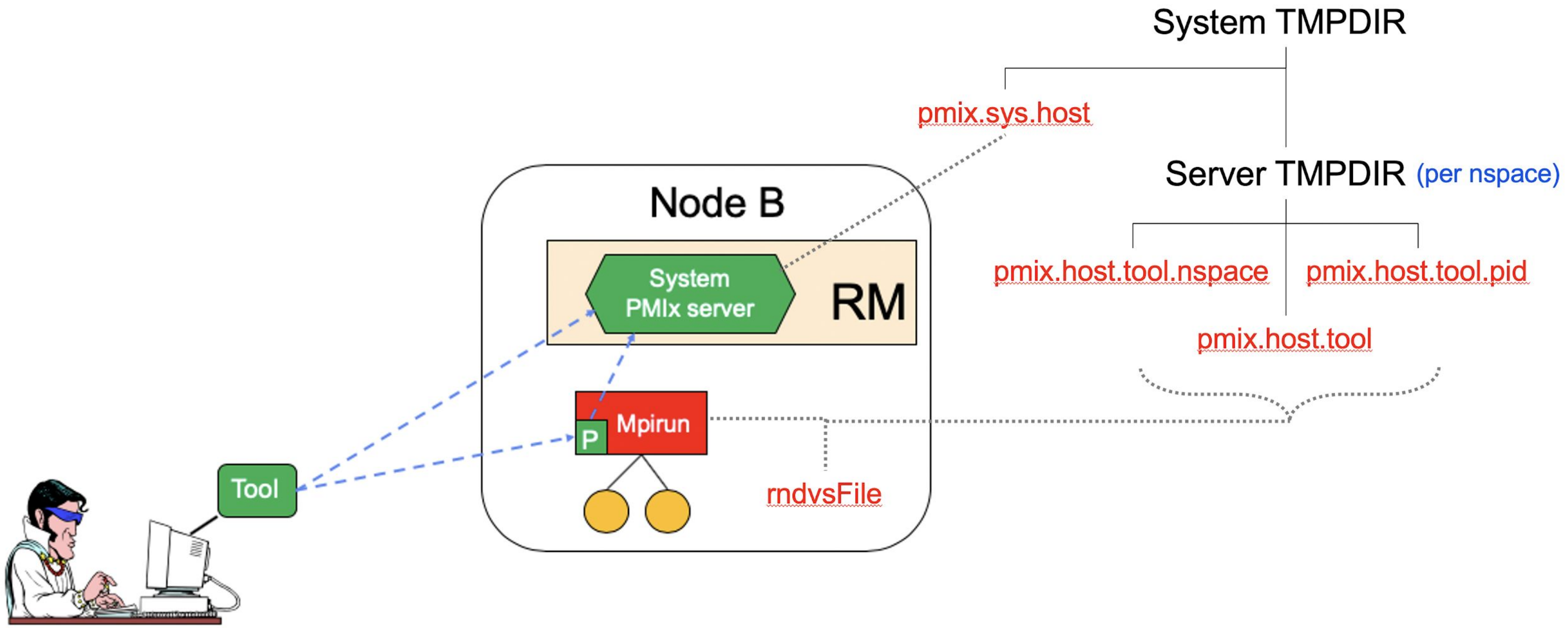


PMIx Launch Orchestration



*RM daemon, mpirun-daemon, etc.

Tool Support File Layout



Current Support (I)

- Typical startup operations
  - Put, get, commit, barrier, spawn, [dis]connect, publish/lookup
- Tool connections
  - Debugger, job submission, query, forward stdio
- Generalized query support
  - Job status, layout, system data, resource availability
- Event notification
  - App, system generated
  - Subscribe, chained
  - Preemption, failures, timeout warning, …
- Logging
  - Status reports, error output
- Flexible allocations
  - Release resources, request resources

Current Support (II)

- Network support
  - Security keys, pre-spawn local driver setup
- Obsolescence protection
  - Automatic cross-version compatibility
  - Container support
- Job control
  - Pause, kill, signal, heartbeat, resilience support (C/R coordination)
- Async definition of process groups
  - Rolling startup/teardown

Programming Model Support

- Open MPI
  - Harvest/forward “OMPI_*” (customize with MCA param)
  - Read/forward default MCA params from system, user files
  - Setup OMPI-specific envars (MPI-3), OMPI-required job-level params
- OpenSHMEM
  - Harvest/forward “SHMEM_*”, “SMA_*” (customize with MCA param)
- Hybrid
  - Intra-process, async event notification between programming models
  - Autodetect hybrid model operation
  - Negotiate resource contention using intra-process events

OpenPMIx Architecture

- MCA Component Architecture
  - “Borrowed” from Open MPI
  - Same build system
- Dependencies
  - Required: libevent or libev, HWLOC
  - Optional: curl, libjansson (2.11 or higher), Cython (Python bindings), lustre
  - Autodetect: zlib
- Key frameworks
  - Ptl: client/server communication
    - TCP, usock (deprecated) components
    - Rendezvous files written in system tmpdir, session tmpdir
  - Gds: key-value data store
    - Hash (always on), ds12/ds21 shared memory

No Embedded Libraries!

https://openpmix.github.io/code/getting-the-reference-implementation

Build Tips

- External PMIx for Open MPI
  - Must also use external libevent, HWLOC
  - Must use the *same* libraries for PMIx as for Open MPI
- Direct link for applications
  - PMIx can be called directly from application
  - PMIx_Init is reference counted
  - If also using Open MPI…
    - Embedded PMIx - symbols are exposed, no further linkage required
    - External PMIx - must link to PMIx and external libevent/HWLOC
    - Open MPI wrapper compiler will do the right thing!
- PMIx wrapper compiler (pmixcc)
  - For non-Open MPI apps using PMIx

PMIx Tools

- pattrs : reports the supported attributes
  - Client, server, and host levels
  - Attribute and description provided
- pevent : inject a PMIx event into the system
  - Specify targets and event
- plookup : perform PMIx_Lookup for specified key
  - Accesses system server
- pmix_info : reports build information (ala mpi_info)
- pps : generates report on what jobs are running in system
- pquery : queries system for specified info
  - Request report on storage system capabilities, network condition, etc.
- pmixcc : wrapper compiler for PMIx-based apps

Slurm/Cray Conflicts

- PMI-1, PMI-2
  - Both environments have their own libraries
- PMIx
  - Provides backward compatibility libraries for PMI-1, PMI-2
  - Allows apps/libs that use PMI-1, PMI-2 to run against PMIx without changes
  - Translates PMI-1, PMI-2 calls to their PMIx equivalent
- Installing to default location can overwrite native libraries!
  - Native PMI libraries and PMIx are not cross-compatible
  - Recommendation - disable build of backward compatibility libraries
    - --disable-pmi-backward-compatibility
    - This will probably become the default as usage has greatly declined

Open MPI: The MPI Stuff

MPI Frameworks (v4.x)

● `bml`:	BTL multipliexing layer
● `btl`**:	Byte transport layer
● `coll`:	MPI collectives
● `fbtl`:	MPI file byte transfer layer
● `fcoll`:	MPI file collectives
● `fs`:	MPI file management
● `hook`:	Generic hooks
● `io`:	MPI IO
● `mtl`:	Matching transport layer
● `op`:	MPI reduction operations
● `osc`:	MPI one sided communications
● `pml`:	MPI point-to-point communications
● `sharedfp`:	MPI shared file pointer operations
● `topo`:	MPI topologies
● `vprotocol`:	Virtual protocol API interposition

MPI Frameworks (v4.x)

● `bml`:	BTL multipliexing layer
● `btl`**:	Byte transport layer
● `coll`:	MPI collectives
● `fbtl`:	MPI file byte transfer layer
● `fcoll`:	MPI file collectives
● `fs`:	MPI file management
● `hook`:	Generic hooks
● `io`:	MPI IO
● `mtl`:	Matching transport layer
● `op`:	MPI reduction operations
● `osc`:	MPI one sided communications
● `pml`:	MPI point-to-point communications
● `sharedfp`:	MPI shared file pointer operations
● `topo`:	MPI topologies
● `vprotocol`:	Virtual protocol API interposition

These are generally the frameworks that end users care about

io: Top-Level MPI File Operations

- MPI APIs such as MPI_FILE_OPEN, MPI_FILE_READ, MPI_FILE_WRITE, … etc.
- Two choices:
    - `ompio`: Open MPI I/O (the default for most cases)
    - `romio`: ROMIO, from Argonne National Labs / MPICH

coll: MPI Collective Operations

- MPI APIs such as MPI_BCAST, MPI_BARRIER, MPI_REDUCE, … etc.
- Which collective algorithm to use is a complex, multi-variate decision
  - Not generally tuned or selected by end users
- Performance improvements coming in v4.1.0
  - Tuning of default algorithm selection
  - New (optional) collective components coming in v4.1.0
  - Represents years of research from the University of Tennessee, Knoxville, USA

pml: Point-to-Point Messaging Layer

- MPI APIs such as MPI_SEND, MPI_RECV, ... etc.

- There are several PMLs to choose from:
  - `ob1`: Multi-device, multi-rail engine
    - Uses BTL components (byte transfer layer)
  - `cm`: Engine for matching network layers
    - Uses MTL components (matching transport layer)
  - `ucx`: Uses the UCX communication library (Unified Communications X)

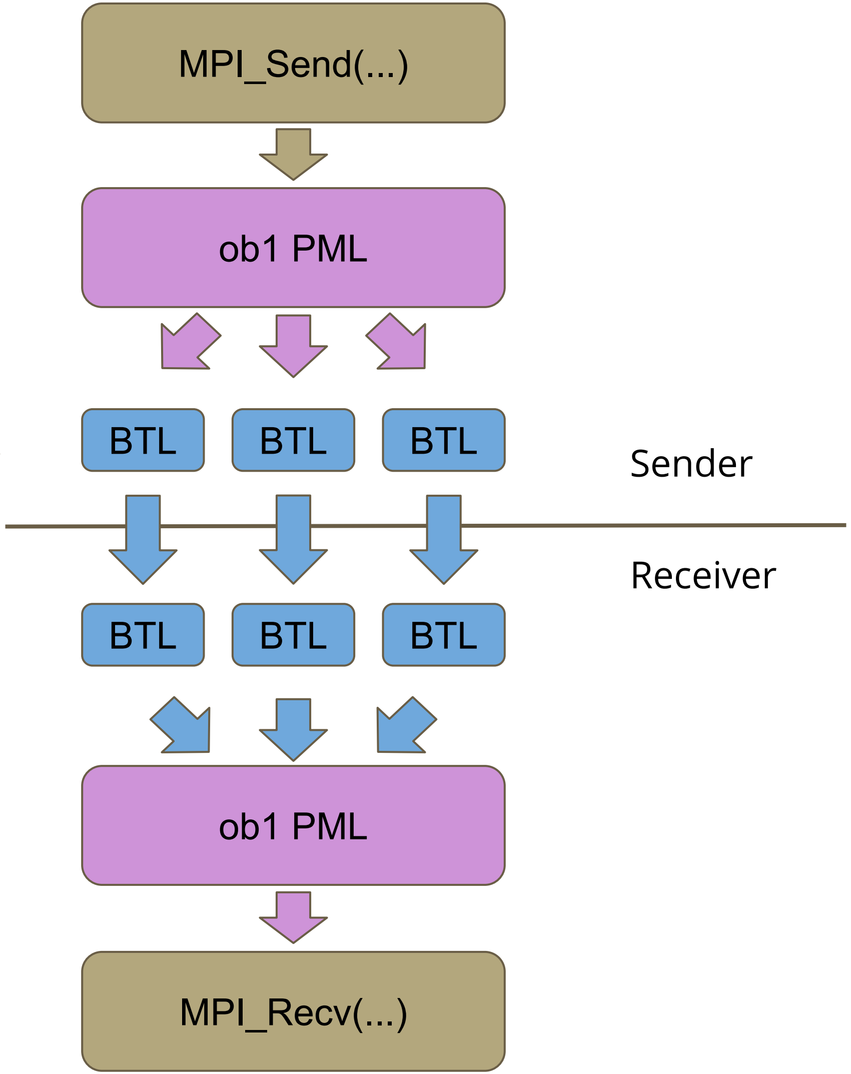
ob1: Multi-Device, Multi-Rail Engine

ob1 will:

1. Pick BTL instance(s) that can reach a given peer
2. Split large messages across relevant BTL instances
3. Re-assemble messages at the receiver

ob1 was Open MPI's original point-to-point transport engine and still works well in many environments.

(yes, "ob1" is a Star Wars reference 😊)



Available BTLs (v4.0.x / v4.1.x)

`ofi`:	Libfabric (OpenFabrics Interfaces)**
`portals4`:	Portals-based networks (uncommon)
`self`:	Process-loopback communications
`sm`:	Shared memory
`smcuda`:	CUDA-aware shared memory
`tcp`:	TCP
`uct`:	UCX**
`ugni`:	Cray uGNI (userspace Generic Network Interface)**
`usnic`:	Cisco usNIC (userspace NIC)

** Denotes a BTL that is not commonly used for MPI point-to-point operations

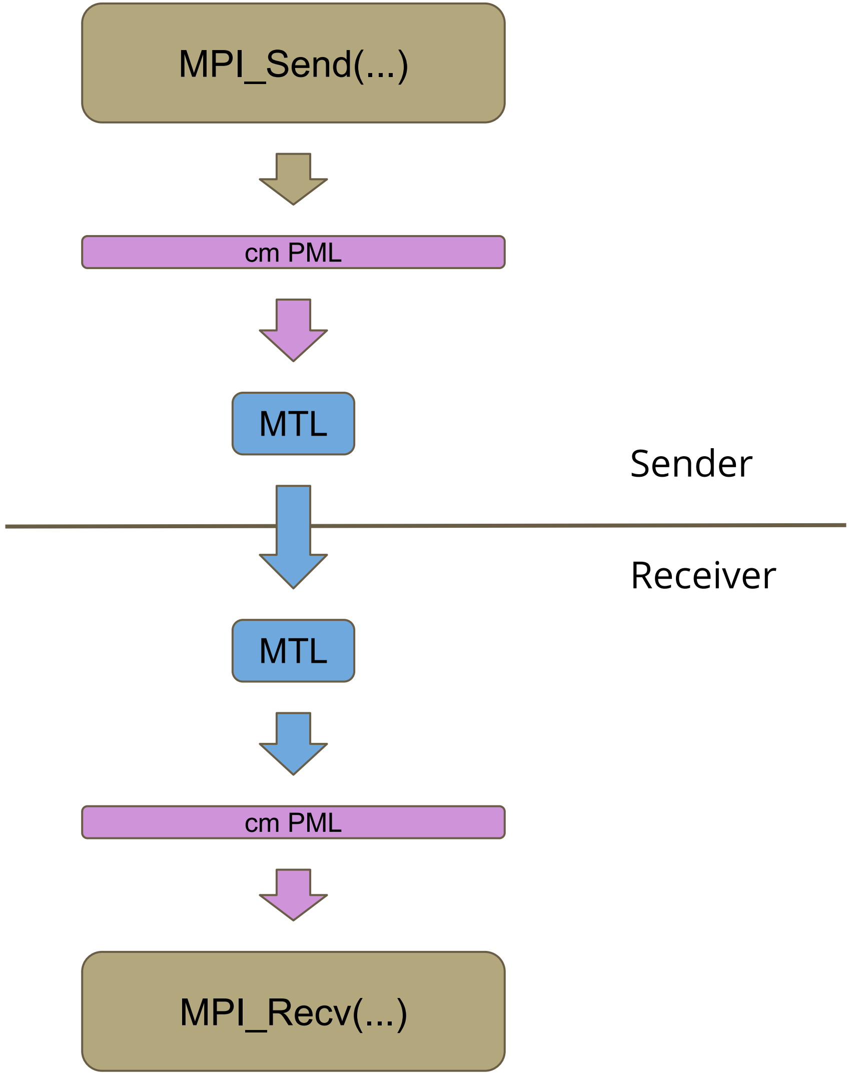
cm: Thin Interface for "Matching" Networks

cm is a thin interface for networks that have "matching" interfaces -- i.e., networks that can natively do MPI-style message matching.

cm uses MTL components (not BTL).

Since network-layer matching is inherently stateful, CM will only use a single MTL in an MPI job (vs. OB1, which can use mutiple BTLs).

cm = a Highlander reference to Connor MacLeod: "there can only be one"



Available MTLs (v4.0.x / v4.1.x)

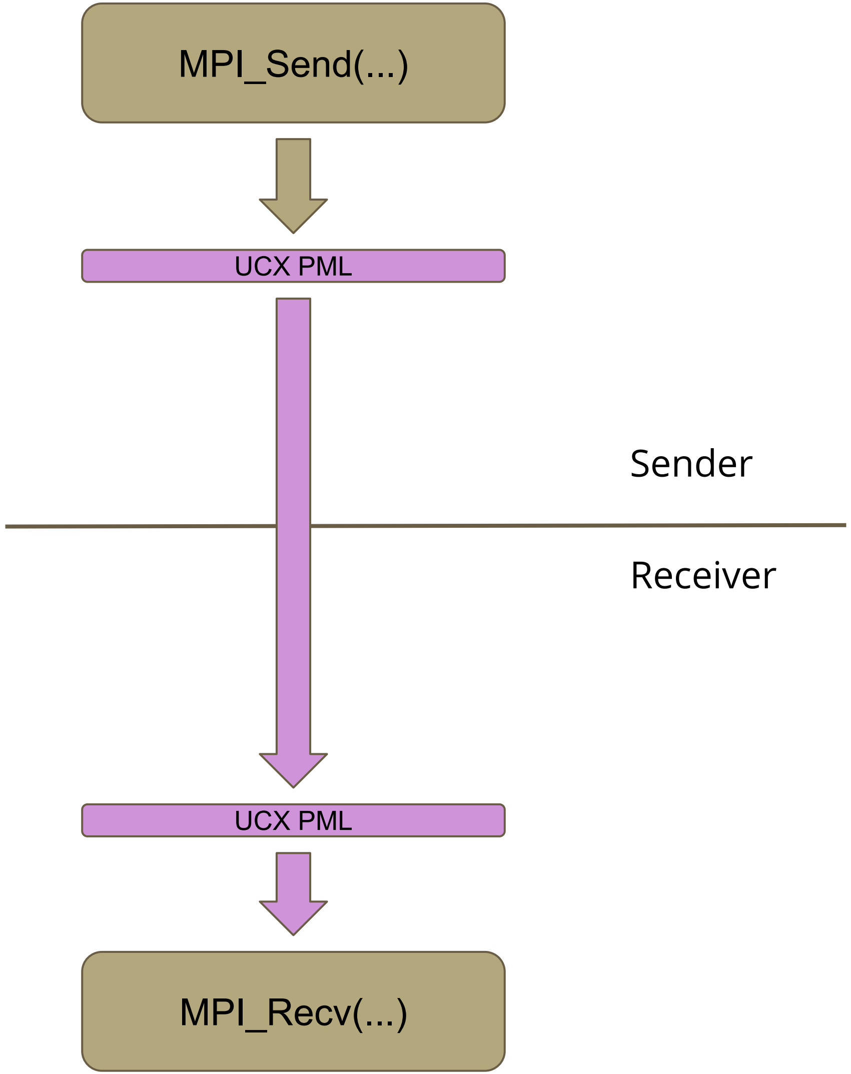
● `ofi`:	Libfabric (OpenFabrics Interfaces)
● `portals4`:	Portals-based networks (uncommon)
● `psm2`:	Single-threaded OmniPath (Performance Scaled Messaging)

ucx: Thin Interface to the UCX Library

UCX is, itself, a multi-device, multi-rail transport library. It has its own engine, and therefore did not need another engine in Open MPI.

Hence, the UCX community decided to write their own (very thin) PML and not use an existing Open MPI engine.

NOTE: The diagram only shows the MPI code blocks (not the UCX library itself).



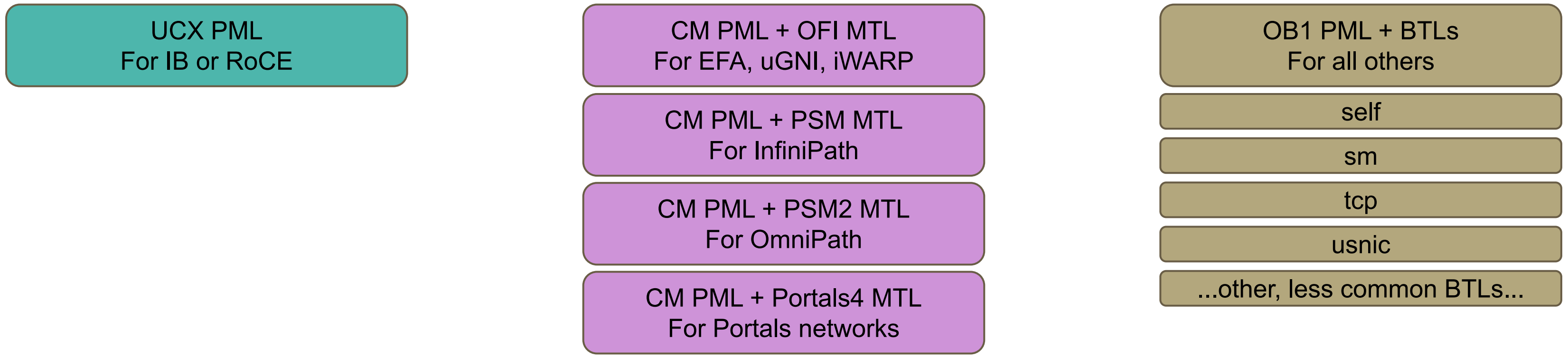
By default, which network gets used at run time?

1. If you have InfiniBand or RoCE, use the UCX PML

2. If you have a matching network or iWARP, use the CM PML + relevant MTL

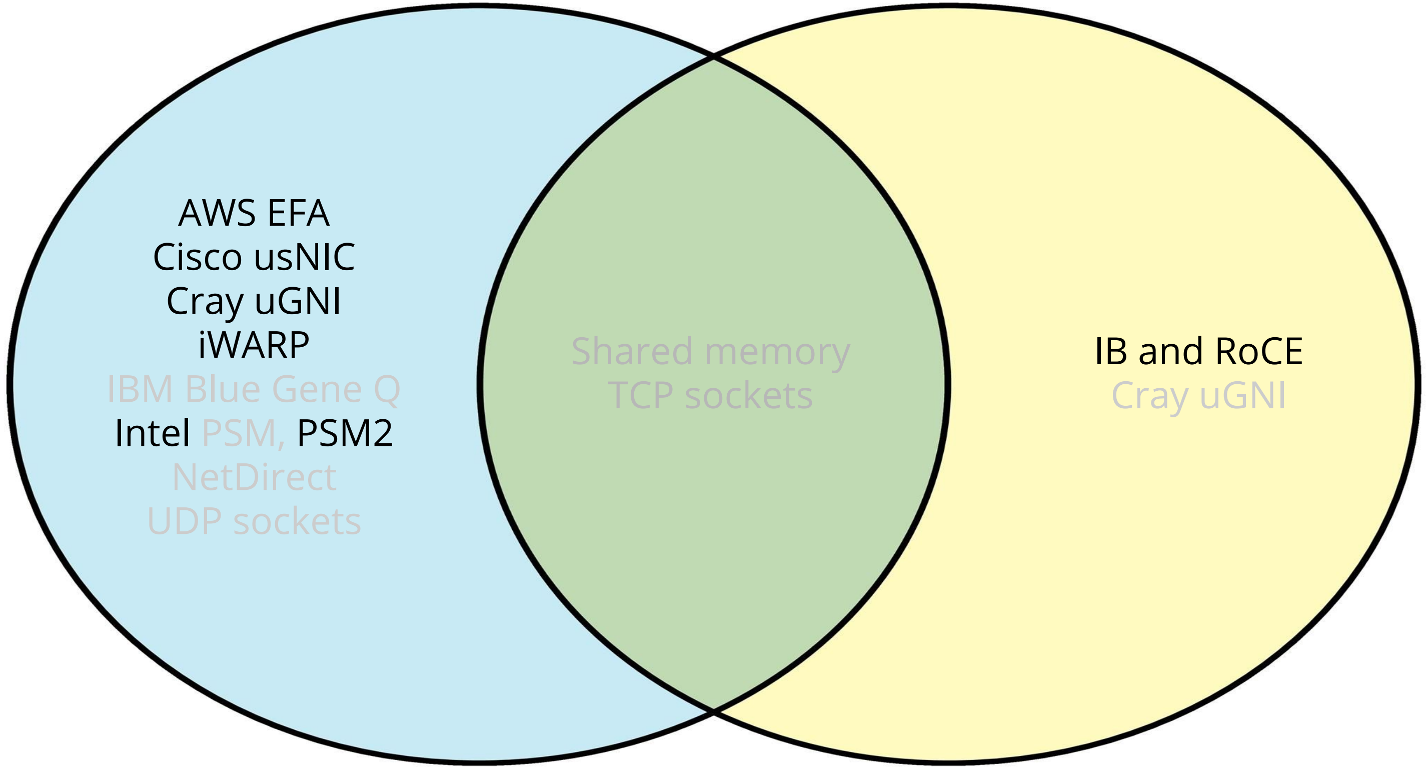
3. Otherwise, use the OB1 PML + appropriate BTLs
   a. Including TCP for “plain” Ethernet environments
   b. Including shared memory (even for single-node runs, such as on laptops)

By default, which network gets used at run time?



Libfabric

UCX



Black transports are used by Open MPI from this library

Grey transports are **NOT** used by Open MPI from this library

What if I want to use a different network stack?

- Force the use of OB1 and BTLs:
  - `mpirun --mca pml ob1 --mca btl [comma-delimited list] …`
- Force the use of CM and MTLs:
  - `mpirun --mca pml cm --mca mtl [MTL] …`
- Force the use of the UCX PML:
  - `mpirun --mca pml ucx …`

It is harmless (but useless) to specify BTLs with cm or MTLs with ob1

CUDA

- UCX and PSM2 support GPUDirect RDMA
  - There are many options and tunable knobs
- Common question: can I directly write CUDA code in my MPI application?
  - Yes, but it is not for the meek
    (i.e., it is complicated to do -- only advised for experts)

- See the Open MPI FAQ “Running CUDA-aware” section for details

CUDA

- Can I run a CUDA-built Open MPI on a node with no GPUs?
  - More specifically: on a node with no CUDA libraries?
- In general: it is easiest if you have the CUDA libraries installed
- Specifically: with UCX: no
  - You must have the CUDA libraries installed
  - But UCX should gracefully handle not having any GPUs present

Interfacing External Libraries

- Open MPI has a standardized parameter system (“MCA”)

- But many of Open MPI’s components are simply “glue” to external libraries (e.g., Libfabric and UCX)
    - Some of these components utilize Open MPI MCA parameters
    - Others do not
    - Example: the UCX PML is controlled via UCX-specific environment variables

Interfacing External Libraries

- Can set MCA parameters via:
    - Command line
        - `mpirun --mca foo bar --mca baz yow ...`
    - Environment variables
        - `export OMPI_MCA_foo=bar`
    - Text config files
        - INI-style key=value text file
        - `$PREFIX/etc/openmpi-mca-params.conf`
            - ...more on this file in part 3

Interfacing External Libraries

- The ompi_info command can show you all available MCA params
    - `ompi_info --all [--parsable]`
    - These are exactly Open MPI's MPI_T control variables

- "Level" of MCA param:
    - Level 1: End user / basic
    - Level 2: End user / detailed
    - Level 3: End user / all
    - Level 4: Application tuner / basic
    - Level 5: Application tuner / details
    - Level 6: Application tuner / all
    - Level 7: MPI developer / basic
    - Level 8: MPI developer / details
    - Level 9: MPI developer / all

Questions?

That's it for part 2!

Join us for part 3 in four weeks:
August 5, 2020
8am US Pacific / 11am US Eastern / 3pm UTC / 5pm CEST

Thank you!

Join us for part 3 in four weeks:

August 5, 2020

8am US Pacific / 11am US Eastern / 3pm UTC / 5pm CEST