Key-Value Exchange Modes: Put/Commit/{Fence}/Get Semantics

Joshua Hursey (IBM)

David Solt (IBM)

Two Models for Key-Value Pair Exchange in PMIx

- **Primary means of exchanging data in PMIx is via key-value pairs (KVPs)**

- **Process Related Key-Value Exchange:**
  - PMIx_Put / PMIx_Commit / PMIx_Fence(COLLECT) / PMIx_Get
  - PMIx_Put / PMIx_Commit / PMIx_Fence(Sync) / PMIx_Get
  - PMIx_Put / PMIx_Commit / PMIx_Get
  - PMIx_Get (for instant-on environments)

Focus here today

- **Non-Process Related Key-Value Exchange:**
  - PMIx_Publish / PMIx_Lookup / PMIx_Unpublish

Process Related Key-Value Exchange: Overview

- **Four sets of APIs that allow PMIx processes to share key-value pairs:**

PMIx_Put	Create a KVP associated with the calling process
PMIx_Commit	Make all KVPs previously 'put' available to other PMIx processes
PMIx_Fence	Synchronize and, optionally, exchange data between a set of processes
PMIx_Get	Access KVPs

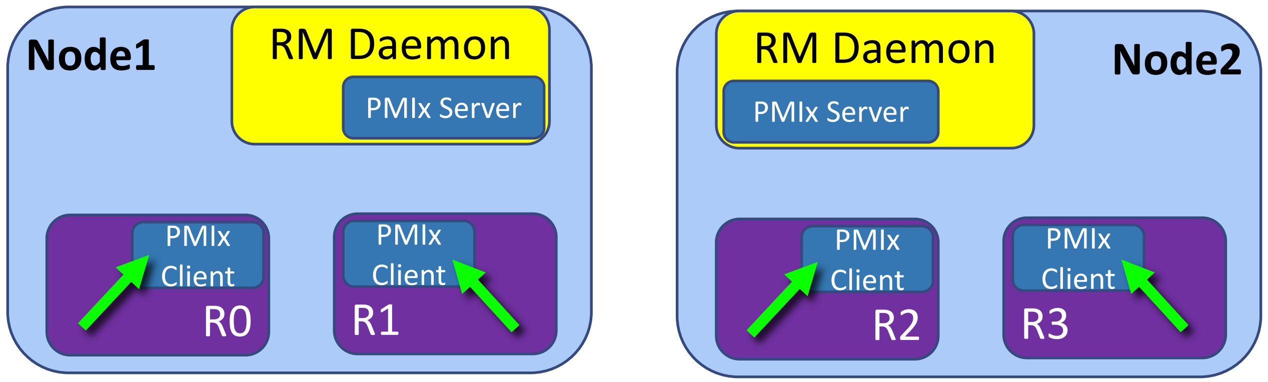
- **Three wireup models:** (*modex* = module exchange = business card exchange)
  - **Instant-On**: Use only **PMIx_Get** to access pre-populated connectivity information from the job-level data. No KVP exchange or synchronization necessary.
  - **Direct Modex**: (Default in PMIx) Data is shared between processes on-demand based on first access to the remote data using **PMIx_Get**.
  - **Full Modex**: (Traditional model) A collective fence operation exchanges all of the committed KVPs to all involved PMIx servers. **PMIx_Get** calls after the fence operation may complete faster at the cost of the data exchange and resulting memory footprint.

PMIx Key-Value Pair Data Realms

- **PMIx Key-Value Pairs (KVPs) exist in one of a few different data realms**
  - User-defined KVPs can only be associated with the process-level data realm
  - KVPs in all other data realms are established by the PMIx Server
- **PMIx KVP Data Realms**
  - **Node-level**: KVPs associated with all processes that share the same node
  - **Session-level**: KVPs associated with the allocated set of resources to this user
  - **Job/Namespace-Level**: KVPs associated with the parallel/distributed job in the session
  - **Application-Level**: KVPs associated with all processes in the job that were launched together with the same binary (or other defined grouping such as argument set)
  - **Process-level**: KVPs associated with a specific process

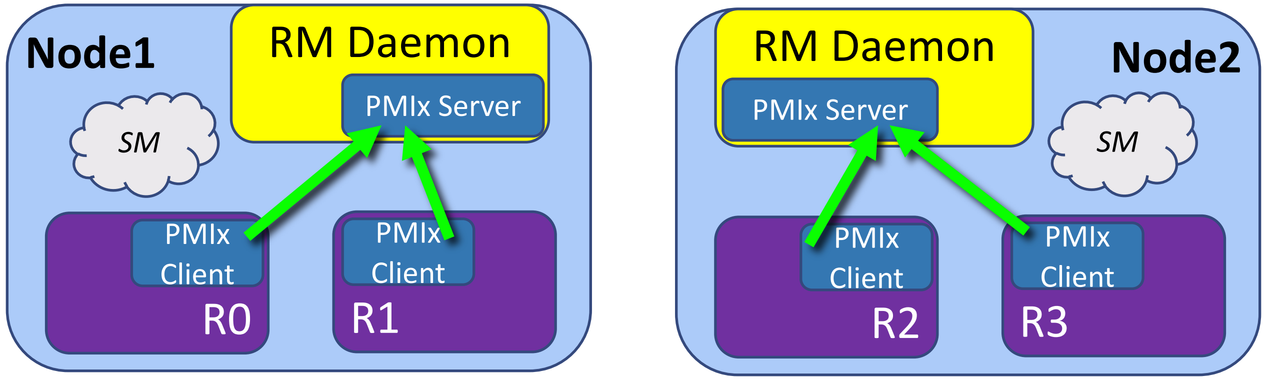
PMIx_Put / PMIx_Store_internal

- **PMIx_Put(scope, key, value)**
  - Prepare a key-value pairs associated with the caller to be shared in the specified scope
    - The caller's namespace & rank are automatically stored with this KVP
    - The KVP is not accessible to other processes until committed
  - scope: **PMIX_LOCAL** (same node only), **PMIX_REMOTE** (remote nodes only), **PMIX_GLOBAL** (everyone), **PMIX_INTERNAL** (this process only)
- **PMIx_Store_internal(proc, key, value)**
  - Store a KVP associated with the specified proc for later access by only the calling process
  - Useful when storing information about a process that was not gathered with PMIx.



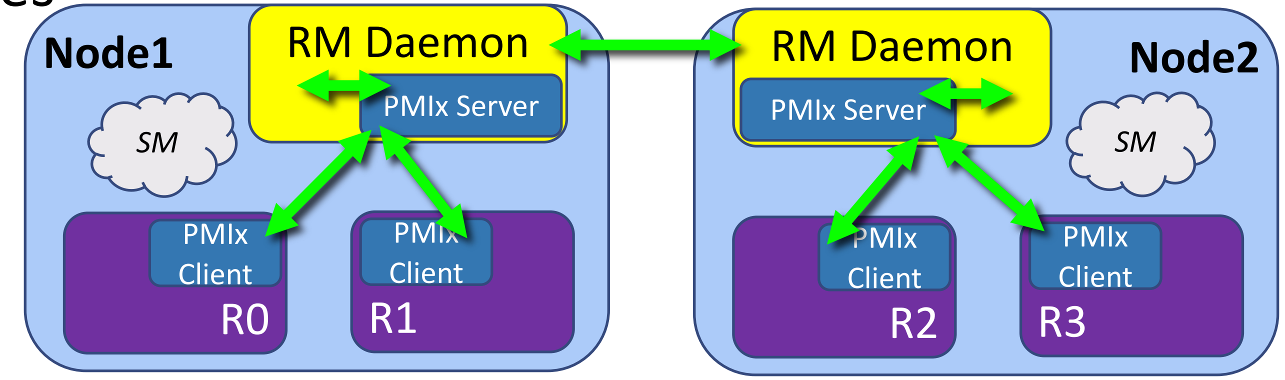
PMIx_Commit

- **PMIx_Commit()**
  - Make key-value pairs previously staged with **PMIx_Put** accessible to other processes
    - Those KVPs with **PMIX_INTERNAL** scope remain cached in the caller-local PMIx client library.
    - Those KVPs with **PMIX_REMOTE** scope are cached only at the PMIx server library and are not accessible to other PMIx clients on the same node.
  - PMIx Client to PMIx Server transmission
    - The transmission of data from the client to server occurs without interrupting the RM Daemon
    - There is (currently) no commit upcall into the RM Daemon hosting the PMIx Server instance
  - The PMIx Server may coordinate with the PMIx clients to create a node-local shared memory segment for fast access to these KVPs.



PMIx_Fence / PMIx_Fence_nb

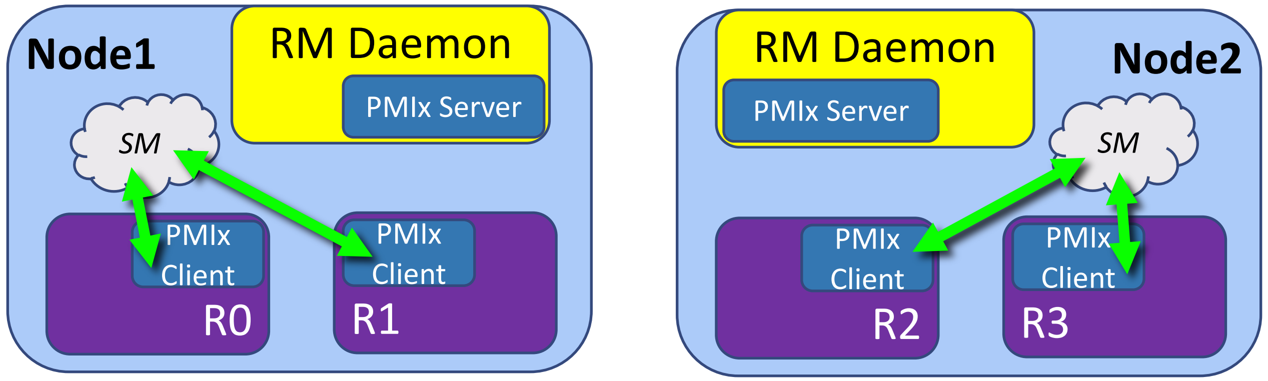
- **PMIx_Fence(procs[], nprocs, info[], ninfo)**
  - Collective barrier operation over the set of processes
    - Wildcard can be used for 'all' processes in the namespace
    - The ordering and content of the proc[] array defines the fence signature used to match between multiple, concurrent fence operations
  - **PMIX_COLLECT_DATA** attribute will request the collection of **PMIX_REMOTE** & **PMIX_GLOBAL** scoped committed KVPs during the collective.
    - The KVPs are then locally available (via **PMIx_Get**) to the designated set of the processes.
    - In MPI terms, this attribute changes the MPI_Barrier into an MPI_Allgatherv operation.
  - Upcall into the RM Daemon (**pmix_server_fencenb_fn_t**) to exchange the data between the involved nodes



PMIx_Get / PMIx_Get_nb

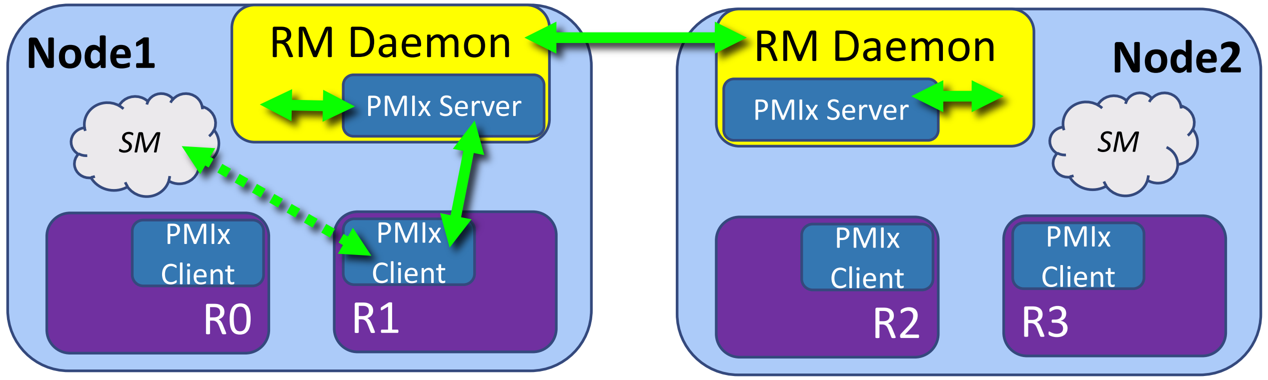
- **PMIx_Get(proc, key, info[], ninfo, value)**
  - Access a key-value pair in the PMIx system
    - The proc and info arguments determine the data realm of the KVP (e.g., session, job, proc)
  - Reserved keys, get will look in the following places for the requested key (in order)
    - Reserved keys are those defined in the PMIx standard (strings prefixed with "pmix")

    1. Local PMIx Client cache
    2. Local PMIx Server cache, if it is for a different namespace
    3. Local PMIx Server cache, if the client asks for a cache refresh (**PMIX_GET_REFRESH_CACHE**)
    4. Return an error (e.g., **PMIX_ERR_NOT_FOUND**)



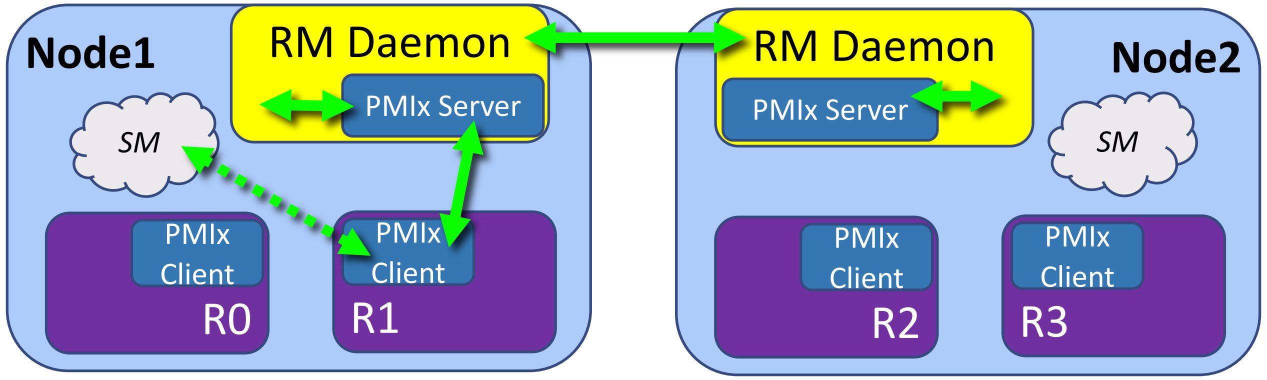
PMIx_Get / PMIx_Get_nb

- **PMIx_Get(proc, key, info[], ninfo, value)**
  - Access a key-value pair in the PMIx system
    - The proc and info arguments determine the data realm of the KVP (e.g., session, job, proc)
  - Non-reserved keys, get will look in the following places for the requested key (in order)
    1. Local PMIx Client cache (**PMIX_OPTIONAL** attribute used to stop search here)
    2. Local PMIx Server cache (**PMIX_IMMEDIATE** attribute used to stop search here)
    3. Target PMIx Server cache (**PMIX_TIMEOUT** attributed used to limit waiting at remote server)
       - If the key is not at the target PMIx Server then **PMIx_Get** will access the currently committed set of values - possibly excluding the KVP requested if it was not yet committed
       - **PMIX_REQUIRED_KEY** attribute used to pass the key being waited upon to the RM daemon in the **pmix_server_dmodex_req_fn_t** upcall so the target will block until the key is available (or a timeout).



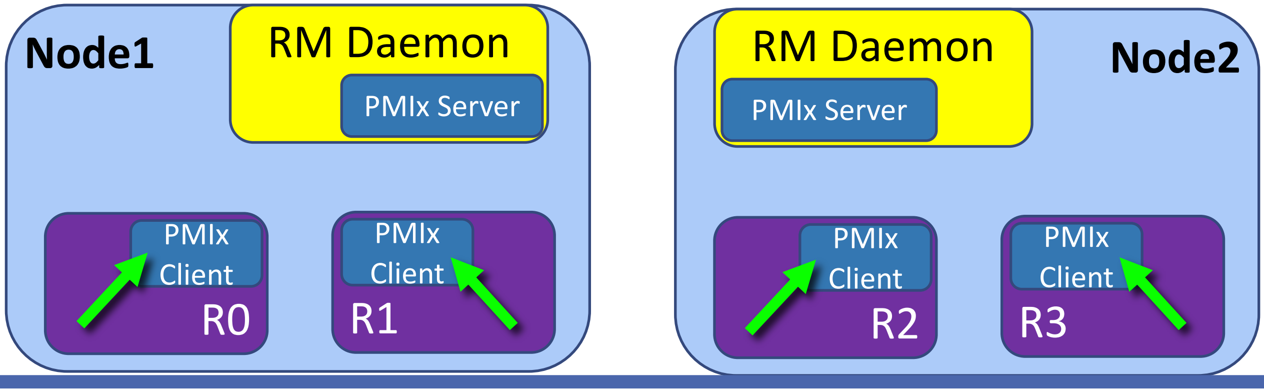
PMIx_Get / PMIx_Get_nb

- **PMIx_Get(proc, key, info[], ninfo, value)**
  - In a **Direct Modex** (or if the key is not available locally), the local and target RM daemons exchange the committed KVPs on-demand. So a **PMIx_Get** could result in an RPC call.
    1. **Node1**: RM daemon gets the **pmix_server_dmodex_req_fn_t** upcall requesting KVPs for a proc
    2. **Node1**: RM daemon determines that it needs to contact Node2 for the data and sends a request
    3. **Node2**: RM daemon calls **PMIx_server_dmodex_request** to access the requested KVP packet
    4. **Node2**: RM daemon sends the KVP data packet to Node1
    5. **Node1**: RM daemon completes the **dmodex_req** callback with the KVP data packet
    6. **Node1**: PMIx server library makes this data available to the local PMIx clients
  - In a **Full Modex**, the committed KVPs are exchanged during the fence so the **PMIx_Get** will *likely* resolve the key from the local process/server cache (often in shared memory).

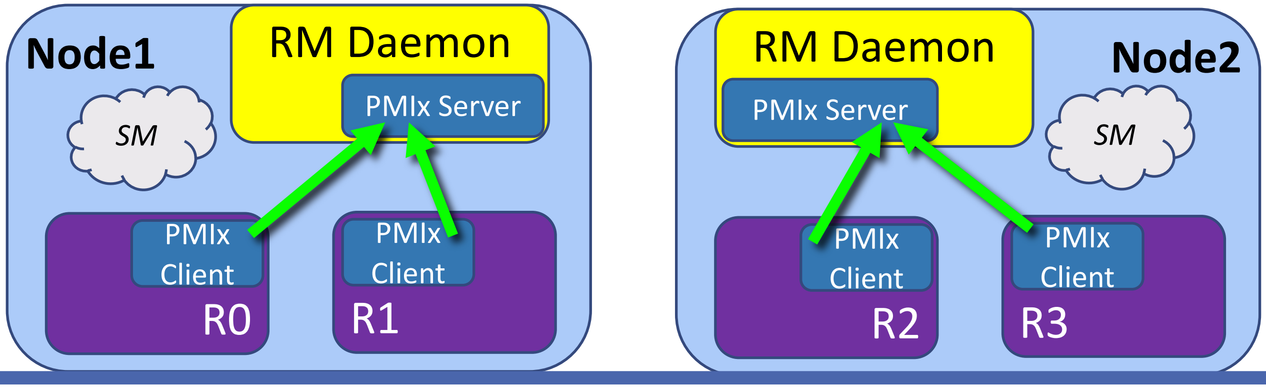


Key-Value Exchange Modes: Put/Commit/{Fence}/Get Semantics

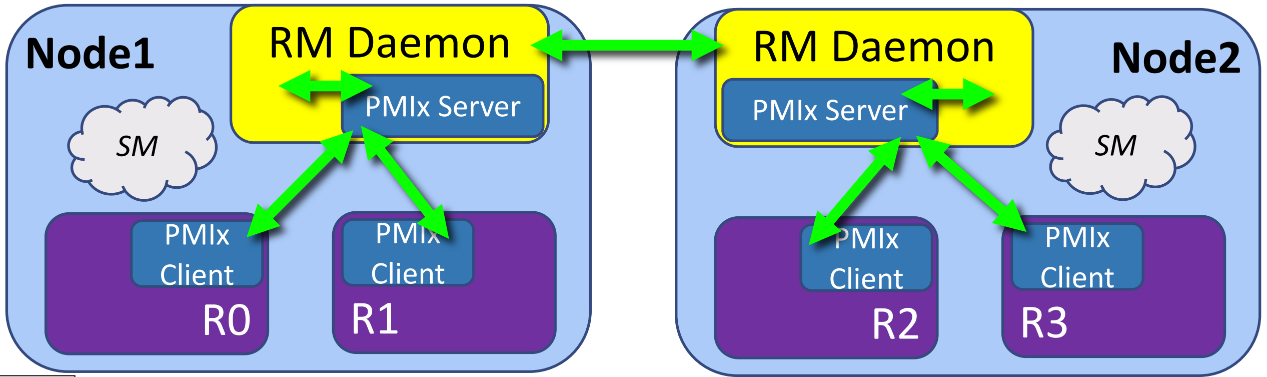
1. PMIx_Put



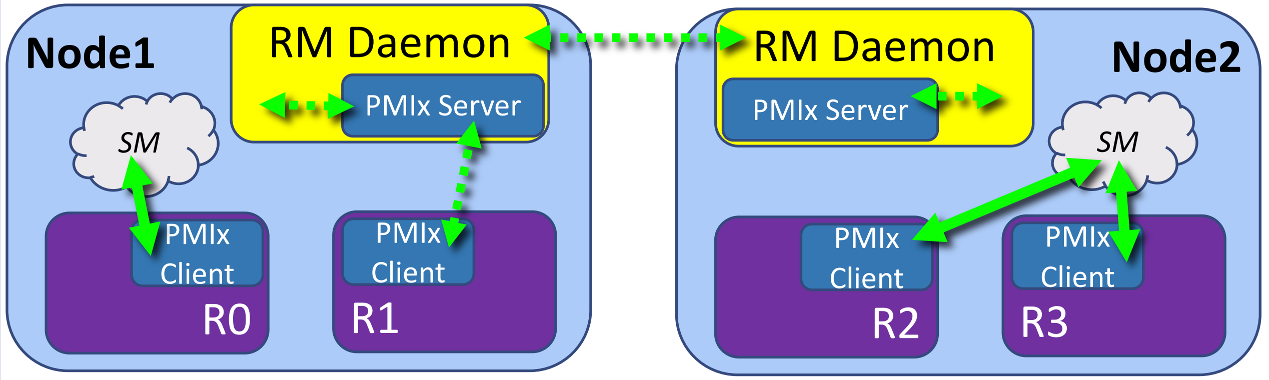
2. PMIx_Commit



3. PMIx_Fence



4. PMIx_Get


Node1
SM
RM Daemon
PMIx Server
PMIx Client
R0
PMIx Client
R1
Node2
RM Daemon
PMIx Server
SM
PMIx Client
R2
PMIx Client
R3