UCF Unified Communication Framework

Welcome to the UCX Tutorial!

Gilad Shainer

HOTI 2021

MISSION: Collaboration between industry, laboratories, and academia to create production grade communication frameworks and open standards for data centric, ML/AI, and high-performance applications

Projects & Working Groups

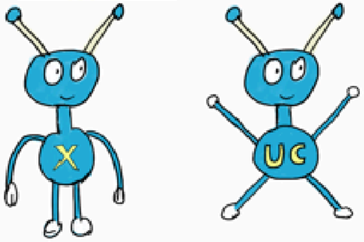
- **UCX – Unified Communication X – www.openucx.org**
- SparkUCX – www.sparkucx.org
- OpenSNAPI – Smart NIC Project
- UCC – Collective Library
- UCD – Advanced Datatype Engine
- HPCA Benchmark – Benchmarking Effort

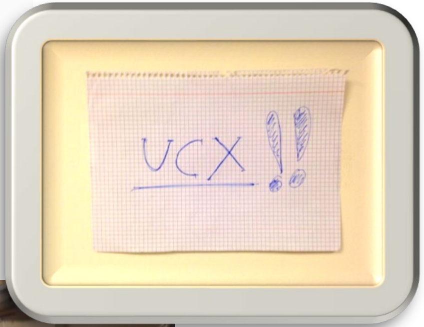
Board members

- **Jeff Kuehn**, UCF Chairman (Los Alamos National Laboratory)
- **Gilad Shainer**, UCF President (NVIDIA)
- **Pavel Shamis**, UCF Treasurer (Arm)
- **Brad Benton**, Board Member (AMD)
- **Yanfei Guo**, Board Member (Argonne National Laboratory)
- **Perry Schmidt**, Board Member (IBM)
- **Dhabaleswar K. (DK) Panda**, Board Member (Ohio State University)
- **Steve Poole**, Board Member (Open Source Software Solutions)

Join https://www.ucfconsortium.org or info@ucfconsortium.org

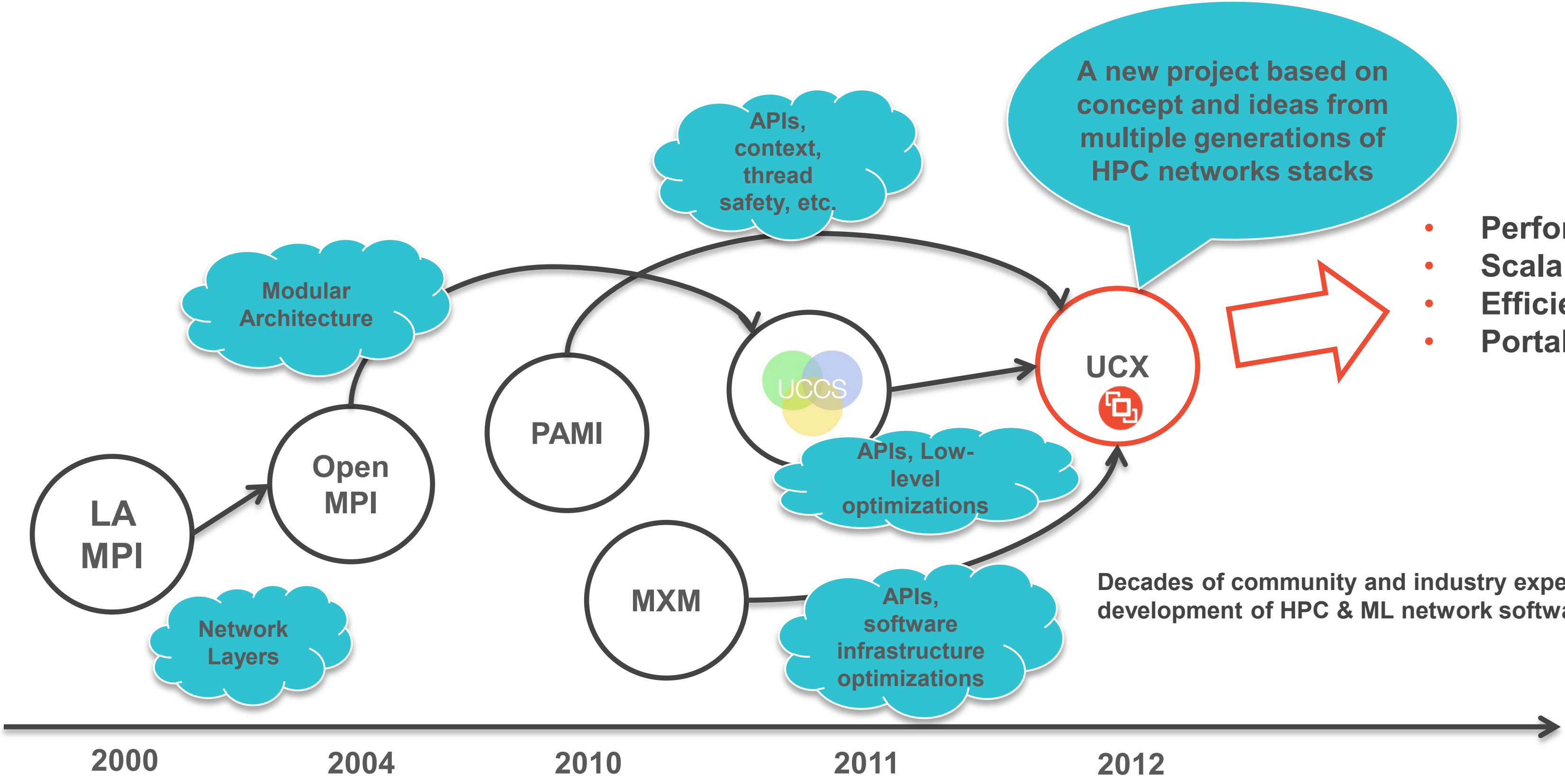
Unified Communication X (UCX)





https://www.hpcwire.com/2018/09/17/ucf-ucx-and-a-car-ride-on-the-road-to-exascale/

UCX – History

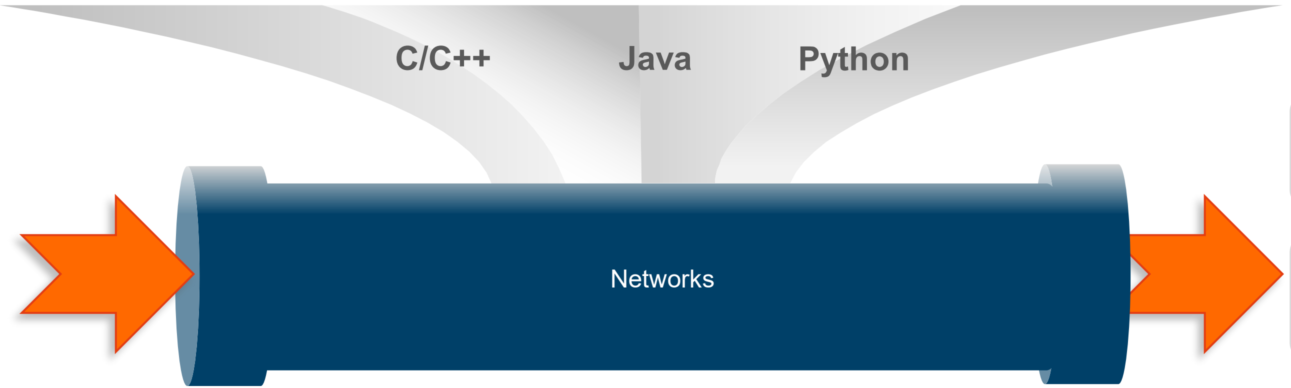


- Performance
- Scalability
- Efficiency
- Portability

Decades of community and industry experience in development of HPC & ML network software

Why UCX ?

MPI | PGAS | ML/AI Frameworks | Data Analytics | Storage | DPU APIs



Host Memory: x86, Arm, Power

GPU Memory: AMD RoCM, NVIDIA CUDA

DPU Memory

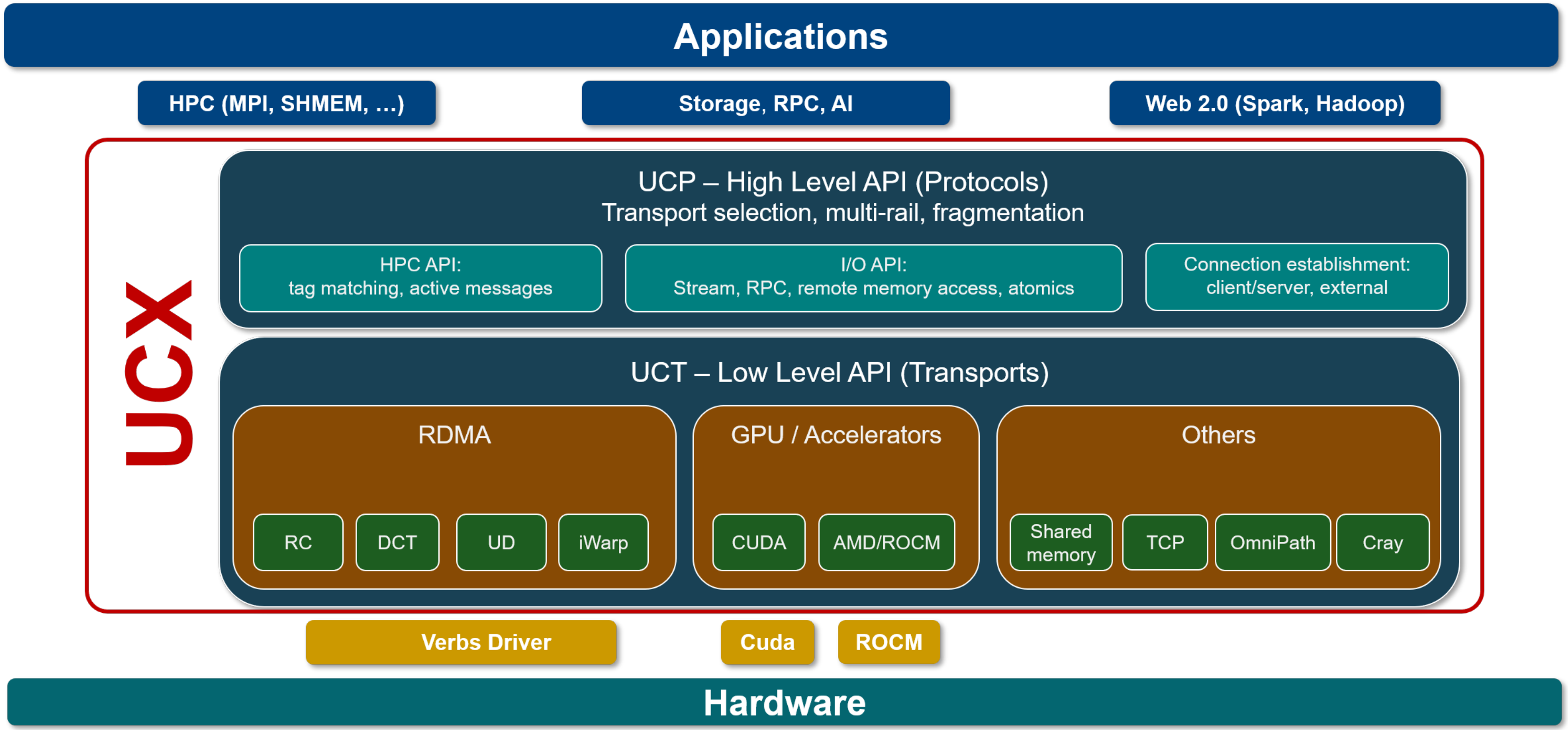
Host Memory: x86, Arm, Power

GPU Memory: AMD RoCM, NVIDIA CUDA

DPU Memory

High-Performance Universal Data Mover

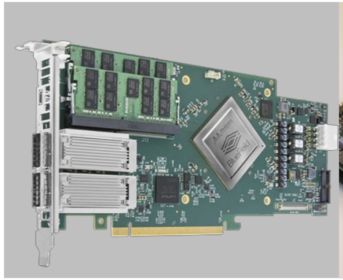
UCX High-level Overview



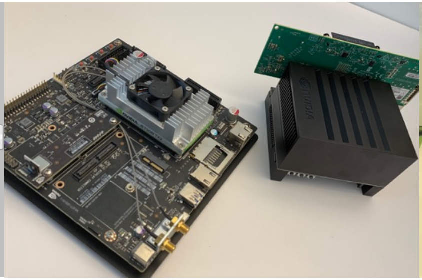
UCX Performance-portability

UCF Unified Communication Framework

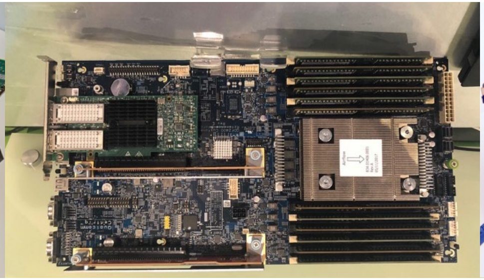
- Support for x86_64, Power 8/9, Arm v8
- U-arch tuned code for Xeon, AMD Rome/Naples, Arm v8 (Cortex-A/N1/ThunderX2/Huawei, Fujitsu A64FX)
- First class support for AMD and NVIDIA GPUs
- Runs on Servers, Raspberry PI like platforms, SmartNIC, Nvidia Jetson platforms, etc.



BlueField DPU

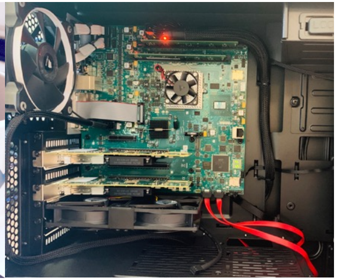


NVIDIA Jetson



Arm ThunderX2

Odroid C2



N1 SDP

UCX Users (Examples)

- MPI implementations: MPICH, Open MPI, NVIDIA HPC-X MPI, Huawei MPI
- PGAS: GasNET
- OpenSHMEM: OSSS SHMEM, Sandia SHMEM, Open MPI SHMEM
- Charm++
- RAPIDS / DASK
- NVIDIA's NCCL

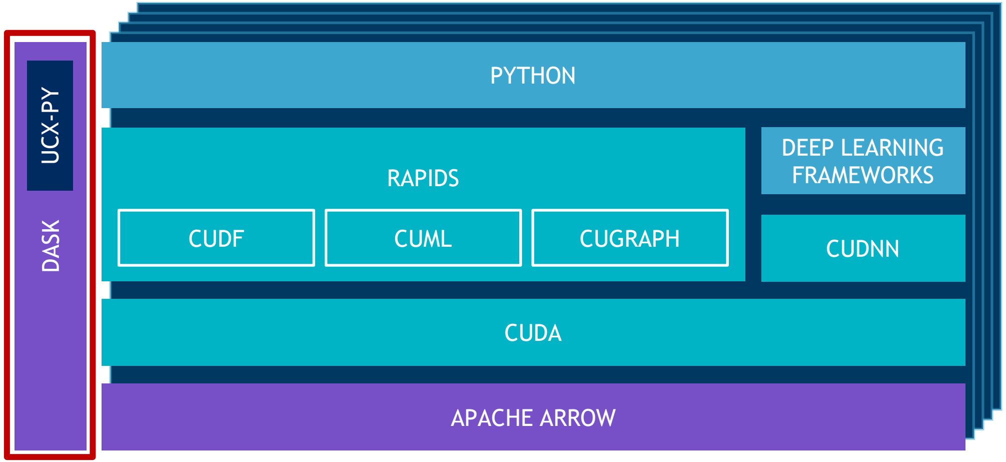


Diagram courtesy of NVIDIA

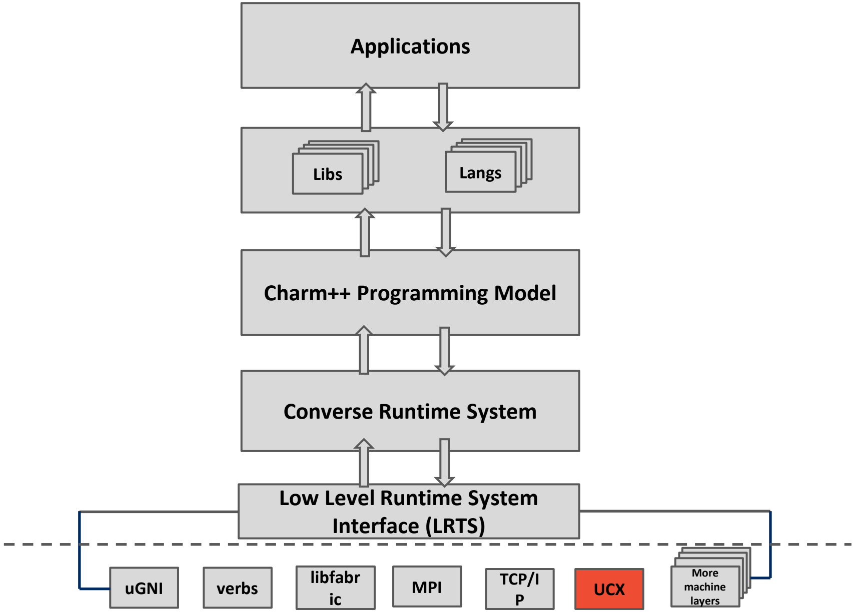
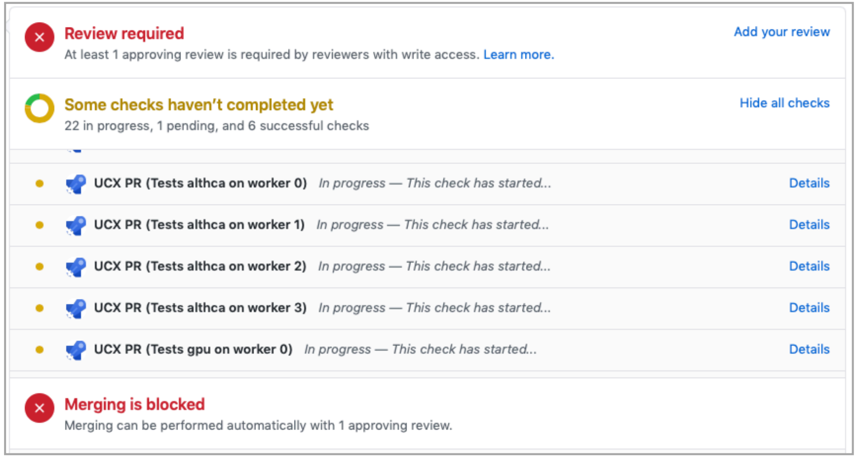
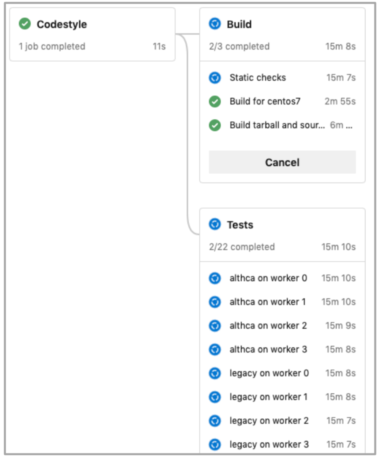


Diagram courtesy of Nitin Bhat @ Charmworks Inc

Over 100,000 tests per commit
220,000 CPU hours per release





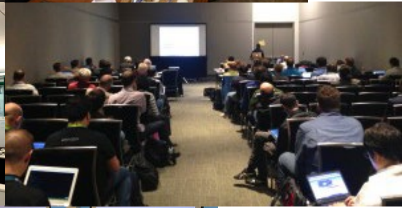
2019 R&D 100 Award

UCX – Useful links

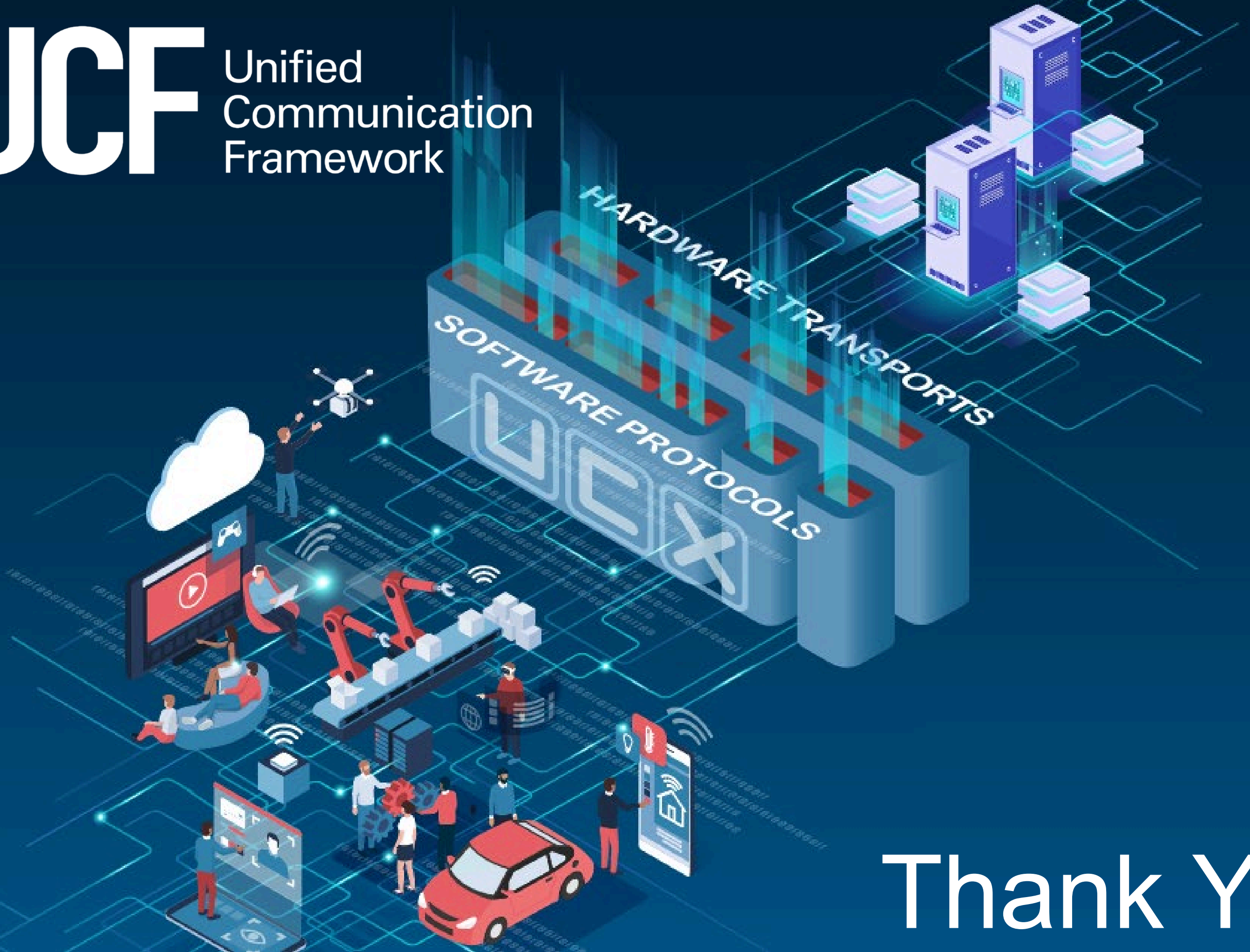
- Code
  - https://github.com/openucx/
- Website
  - www.openucx.com
- Mailing list
  - https://elist.ornl.gov/mailman/listinfo/ucx-group
- Contributor agreement
  - https://www.openucx.org/license/
- User documentation
  - https://openucx.readthedocs.io/







UCF Unified Communication Framework



Thank You

Unified Communication-X (UCX) Tutorial

Jeff Young - Georgia Tech
Gilad Shainer - UCF / NVIDIA
Yossi Itigin, Oscar Hernandez - NVIDIA
Matthew Baker - Oak Ridge National Laboratory

Slide contributors: Pavel Shamis, Alina Sklarevich, Alex Margolis, Swen Boehm, and Oded Paz

Tutorial Material

- Slides and examples

  - For this tutorial - https://github.com/gt-crnch-rg/ucx-tutorial-hoti-21
  - UCX ReadTheDocs - https://openucx.readthedocs.io/

- Slack Channel
  - Join the HOTI Workspace using RDMobile
  - Click on the tutorial RDMobile channel link to join:
    - #tutorial-unified-communication-x-for-performance-portable-network-acceleration

- Q&A
  - UCX mailing list - https://elist.ornl.gov/mailman/listinfo/ucx-group

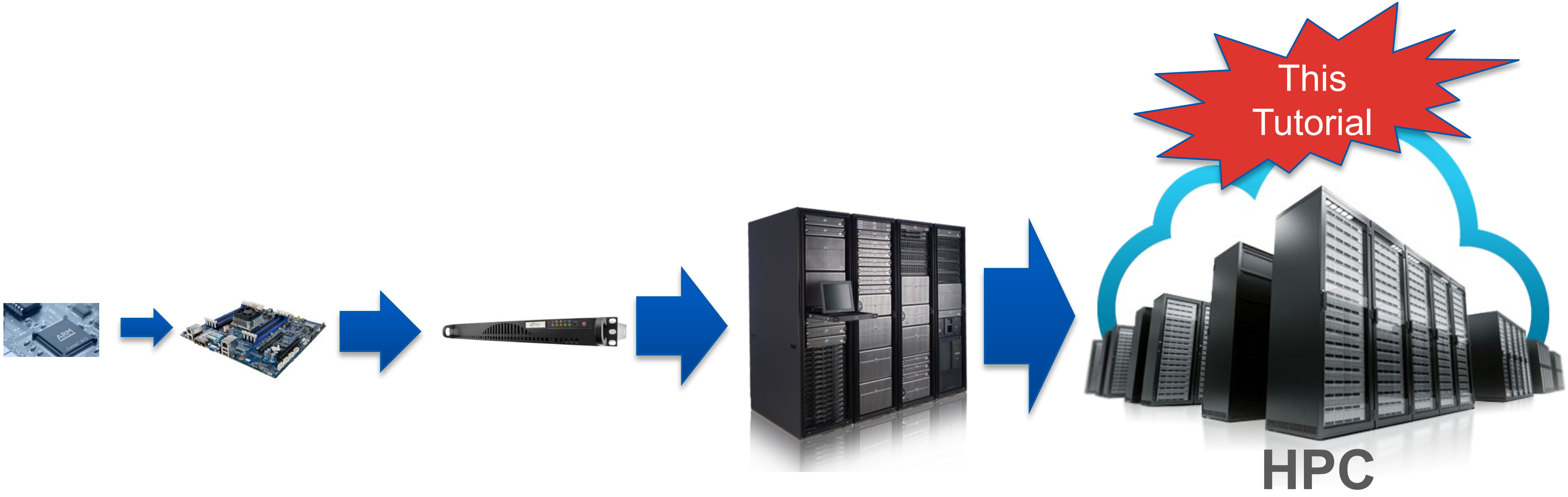
Agenda

- Background: UCF Foundation
  - Overview of existing technologies
  - Unified Communication X Framework
- Introduction to UCX
  - Unified Communication Protocols (UCP)
  - UCX basic examples
- Advanced UCX examples
  - Implementing programming models with UCX

Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview,	Jeff		
	worker and endpoint creation			
10:00 - 10:10	**BREAK**			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	**BREAK**			
11:10 - 11:50	UCX Advanced Topics - bindings,	Matt, Yossi, Oscar		
	OpenMPI and OpenSHMEM integration			

Interconnects

- Interconnects are everywhere: System-on-Chip, chip-to-chip, rack, top-of-the-rack, wide area networks



HPC Example: TCP/IP vs InfiniBand Architecture

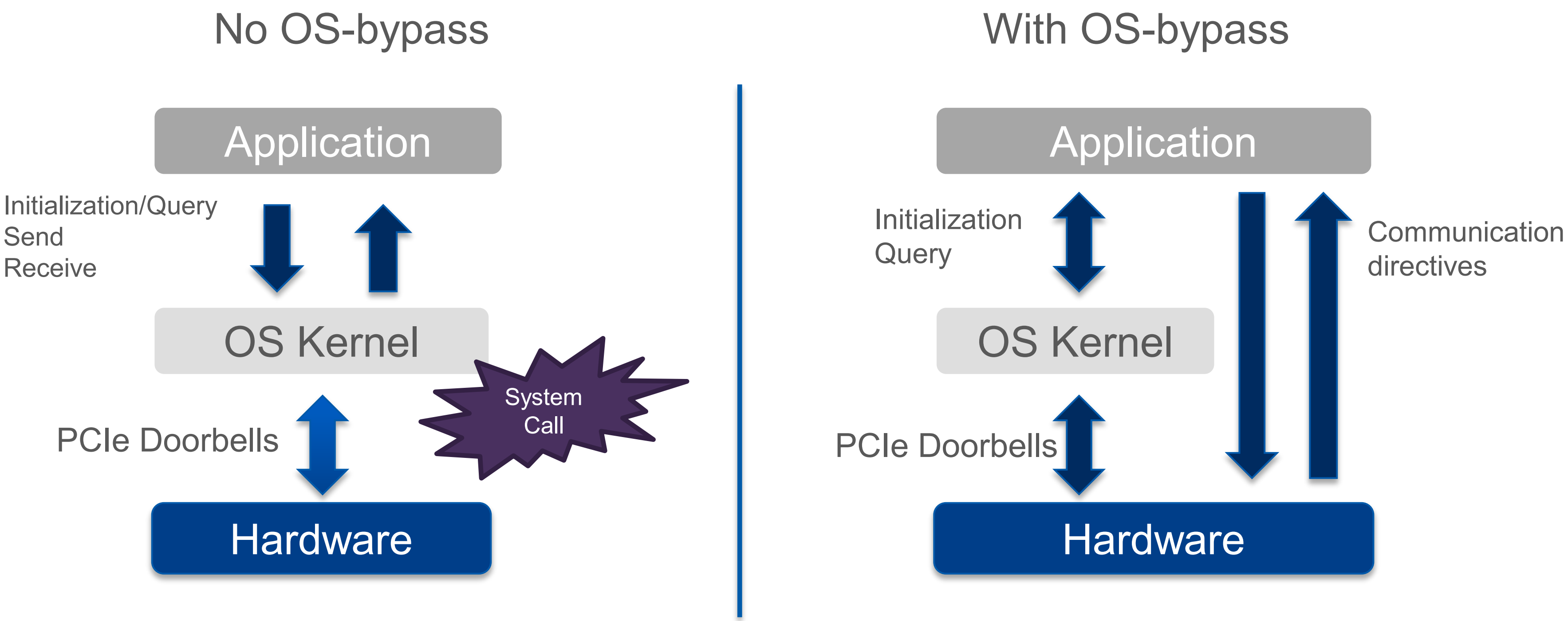
TCP/IP	InfiniBand Architecture	
Application	Upper Layer	Protocols: MPI, NCCL, IP over IB, RDMA
Transport	**Transport Layer**	Hardware-based transports in Host Channel Adaptor (HCA)
Network	**Network Layer**	Protocols to route packets across subnets using routers and global identifiers (GID)
Data Link	**Link Layer**	Local IDs (LID) /subnets, switches forwarding tables (LID/port), flow control, loss less fabric
Physical	**Physical Layer**	How bits are placed in the HW, signaling protocols cables (copper/fiber), etc.

■ Accelerated in Hardware

Key Concepts

- Communication Model
  - Two-sided communication model
    - Send and receive model
  - One-sided communication model
    - Remote memory access and atomics
- Rendezvous
  - Two sides exchange meta-data and use one-sided operations for bulk transfer
- Bypassing the OS
- Transport offload

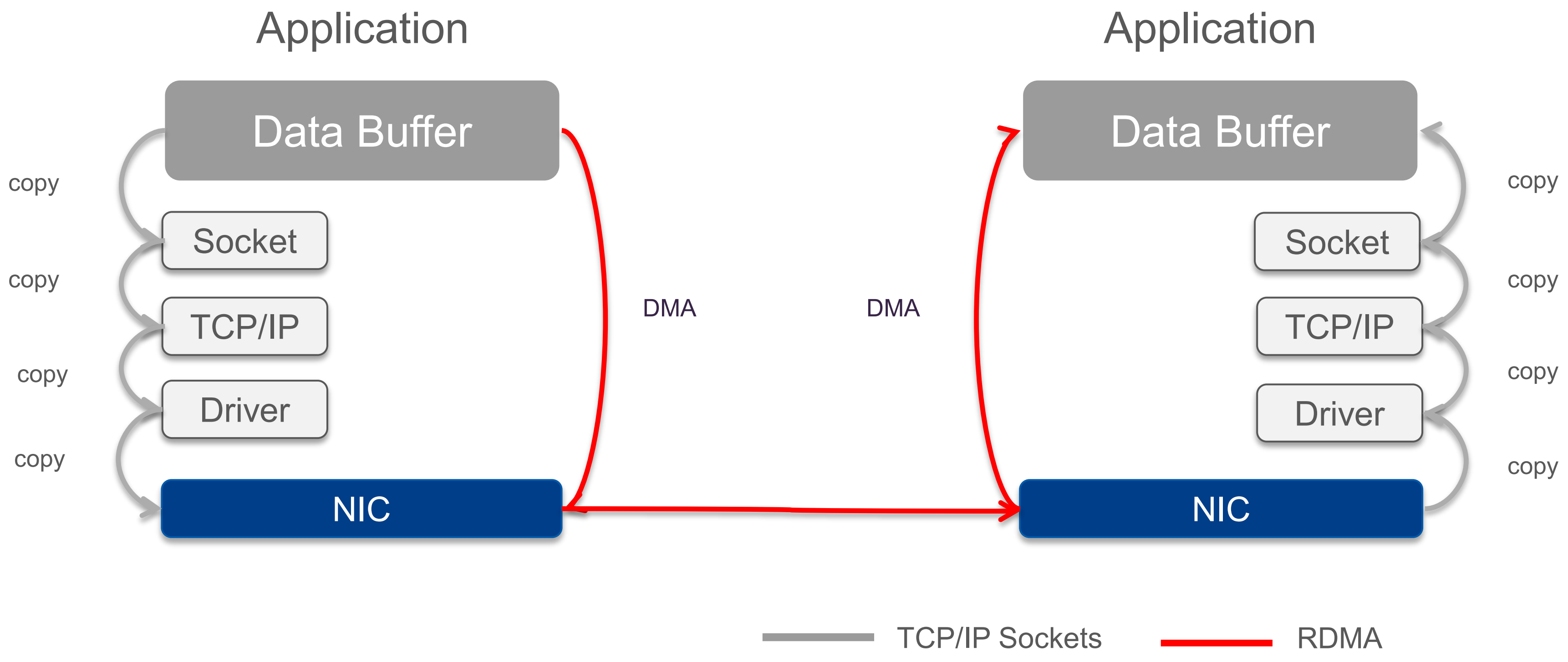
Bypassing the OS



RDMA communication modes

- "Zero" Copy
  - No copies
  - Sender notifies the receiver of its intent
  - Receiver notifies the sender that is ready with destination address
  - Sender sends data to remote memory address
  - Good for large messages
- Buffer Copy
  - Temporary buffers on sender and receivers
  - Additional copies to/from buffers
  - Good for small messages that fit in a buffer

RDMA – “Zero” Copy



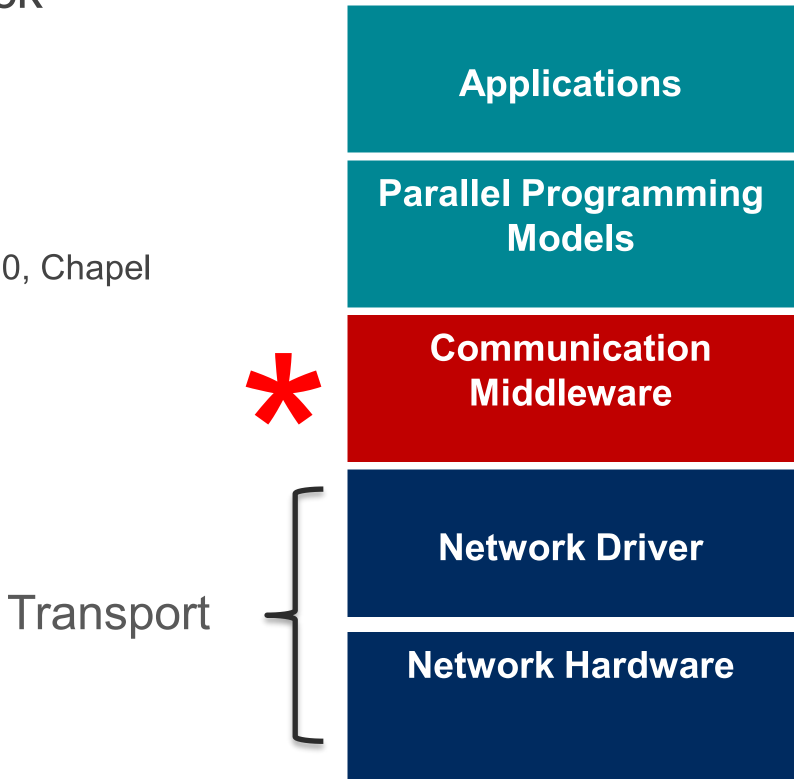
Advanced Communication Semantics

- RDMA Read and Write
- Send / Receive
  - Send / Receive with TAG matching
- Atomic Operations on Remote Memory
  - SWAP
  - CSWAP
  - ADD
  - XOR
- Group Communication directives
  - Reduce, Allreduce, Scatter, Gather, AlltoAll

Socket API:
send() and recv(), or
write() and read(), or
sendto() and recvfrom()

How to improve the HPC Software Stack

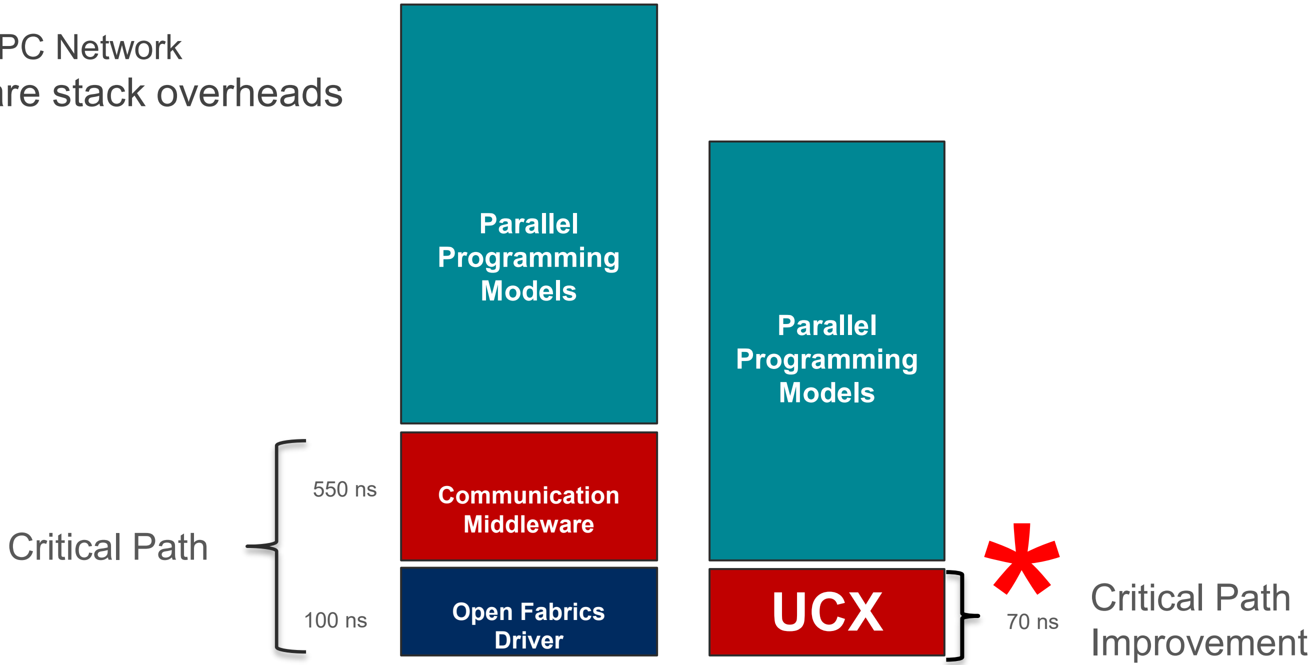
- Applications
  - GROMACS, NAMD, Fluent, Lsdyna, etc.
- Programming models
  - MPI, UPC, OpenSHMEM/SHMEM, Co-array Fortran, X10, Chapel
- Middleware
  - GasNET, MXM, ARMCI, etc.
  - Part of programming model implementation
  - Sometimes “merged” with driver
- Driver
  - OFA Verbs, Cray uGNI, etc.
- Hardware
  - InfiniBand, Cray Slingshot, etc.



How do we improve the network stack?

- Network latency is a key
  - Sub-micro is typical for HPC Network
- Need to eliminate software stack overheads

Example: RMA PUT operation



Network Programming Interfaces (beyond sockets)

- Open Fabric Alliance: Verbs, Udapl, SDP, libfabrics, …
- Research: Portals, CCI, UCCS
- Vendors: Mellanox MXM, Cray uGNI/DMAPP, Intel PSM, Atos Portals, IBM PAMI, OpenMX
- Programming model driven: MVAPICH-X, GasNET, ARMCI
- Enterprise App oriented: OpenDataPlane, DPDK, Accelio

Vendor-Specific APIs

Pros

- Production Quality
- Optimized for Performance
- Support and maintenance

Cons

- Often “vendor” locked
- Optimized for a particular technology
- Co-design lags behind

Open-Source APIs

Pros

- Community (a.k.a. user) driven
- Easy to modify and extend
- Good for research

Cons

- Typically, not as optimized as commercial/vendor software
- Maintenance is challenge

Research API

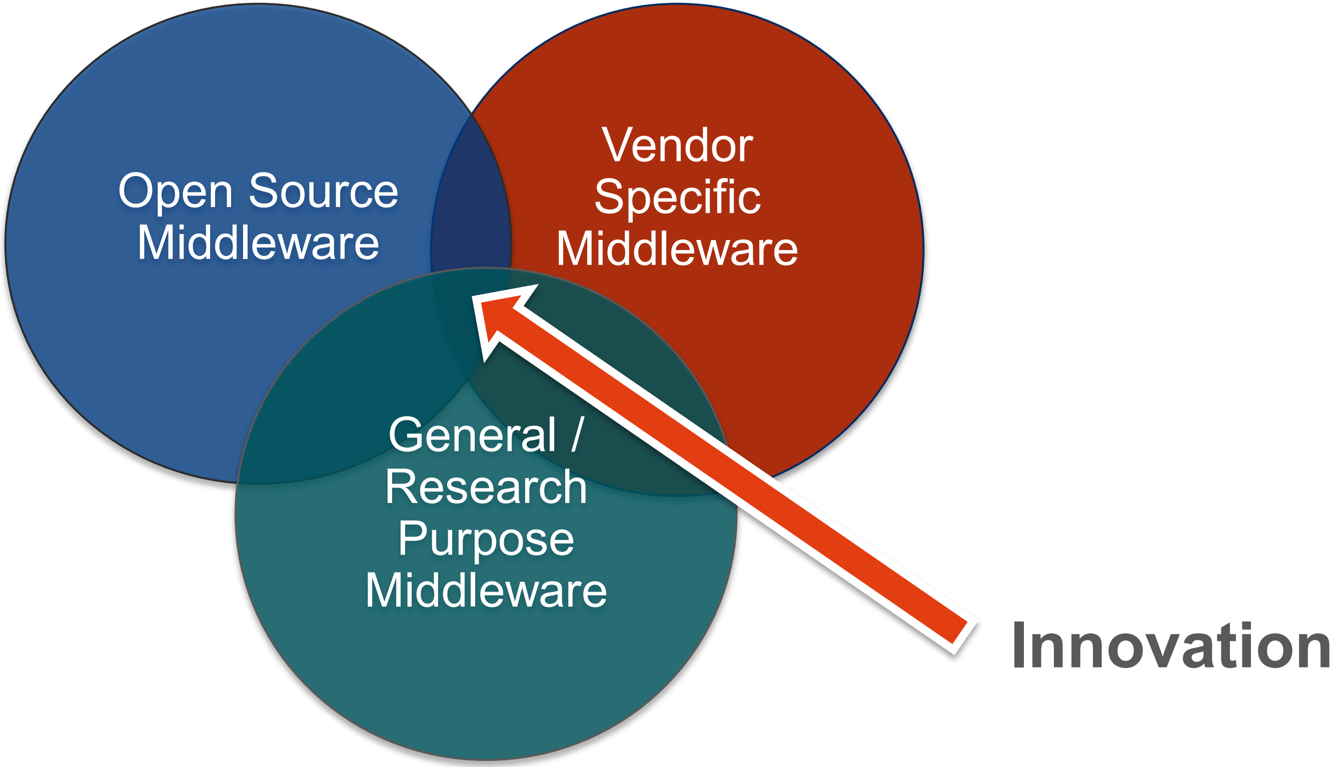
Pros

- Innovative and forward looking
  - A lot of good ideas for “free”

Cons

- Support, support, support
- Typically, narrow focus

Network Programming Interfaces



Unified Communication - X Framework

UCX

UCX Framework Mission

- Collaboration between industry, laboratories, and academia
- Create open-source production grade communication framework for HPC applications
- Enable the highest performance through co-design of software-hardware interfaces
- Unify industry - national laboratories - academia efforts

API

Exposes broad semantics that target data centric and HPC programming models and applications

Performance oriented

Optimization for low-software overheads in communication path allows near native-level performance

Production quality

Developed, maintained, tested, and used by industry and researcher community

Community driven

Collaboration between industry, laboratories, and academia

Research

The framework concepts and ideas are driven by research in academia, laboratories, and industry

Cross platform

Support for InfiniBand, HPE, various shared memory (x86-64, Power, ARM), GPUs

Co-design of Next-Generation Network APIs

A Collaborative Effort

- Mellanox co-designs network API and contributes MXM technology
  - Infrastructure, transport, shared memory, protocols, integration with OpenMPI/SHMEM, MPICH
- ORNL co-designs network API and contributes to the UCCS project
  - InfiniBand optimizations, Cray devices, shared memory
- LANL co-designs network API
- ARM co-designs the network API and contributes optimizations for ARM eco-system
- NVIDIA co-designs high-quality support for GPU devices
  - GPUDirect, GDR copy, etc.
- IBM co-designs network API and contributes ideas and concepts from PAMI
- UH/UTK focus on integration with their research platforms

What’s new about UCX?

- **Simple, consistent, performance portable unified API**

- Choosing between low-level and high-level API allows easy integration with a wide range of applications and middleware.

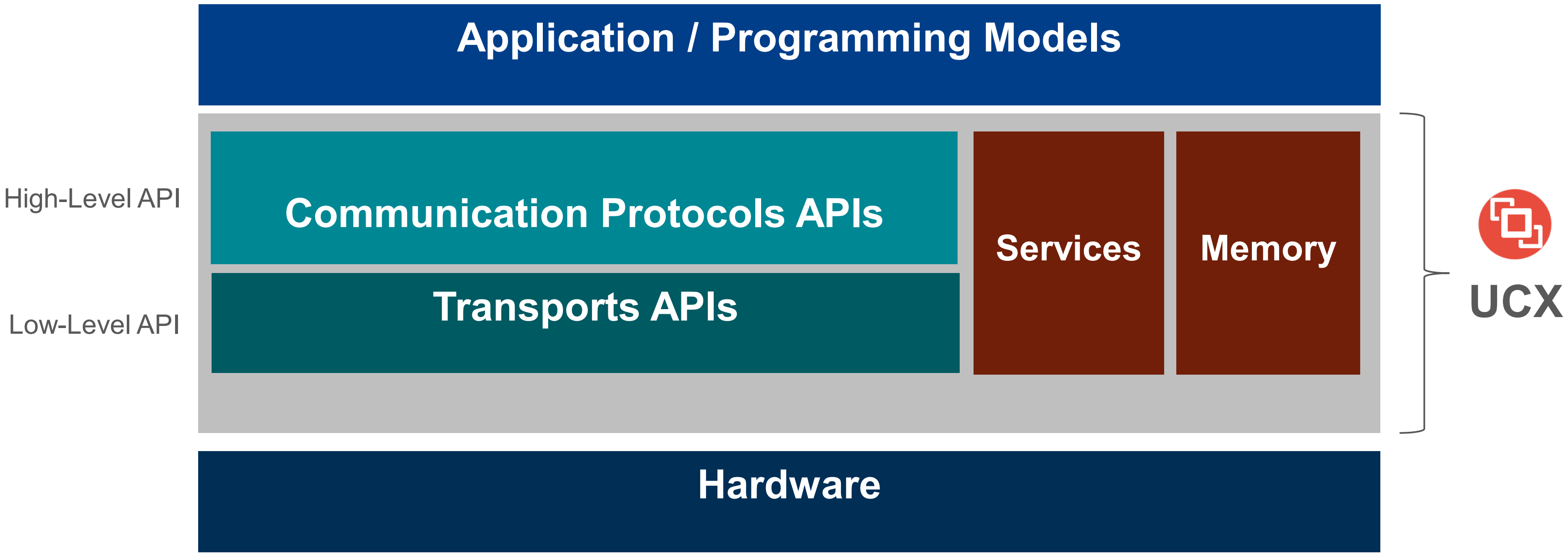
- Protocols and transports are selected by capabilities and performance estimations, rather than hard-coded definitions.

- Support thread contexts and dedicated resources, as well as fine-grained and coarse-grained locking.

- Accelerators are represented as a transport, driven by a generic “glue” layer, which will work with all communication networks.

© 2021 UCF Consortium

Unified Communication X



UCX Framework

UC-P for Protocols

High-level API uses UCT framework to construct protocols commonly found in applications

Functionality:
Multi-rail, device selection, pending queue, rendezvous, tag-matching, software-atomics, etc.

UC-T for Transport

Low-level API that expose basic network operations supported by underlying hardware. Reliable, out-of-order delivery.

Functionality:
Setup and instantiation of communication operations.

UC-S for Services

This framework provides basic infrastructure for component-based programming, data structure support, and useful system utilities

Functionality:
Platform abstractions, data structures support, debug facilities.

UC-M for Memory

This framework provides infrastructure for getting notifications about memory allocate and release events

Functionality:
Platform for memory allocations/dealloc. notifications across devices

Unified Communication X

Applications / Programming Models

High-Level API

UCP - Protocols
Transport selection, multi-rail support, fragmentation
HPC and I/O protocols (tag matching, active messages, RMA, atomics, etc)
Connection establishment (client/server, external)

Low-Level API

UCT - Transports
RDMA (RC, DC, UD, iWarp), Aries (GNI), Accelerators (CUDA ROCm),
Shared Memory (XPMEM, etc), Others (TCP/IP, Omnipath, etc)

UCS - Services

UCM - Memory

UCX

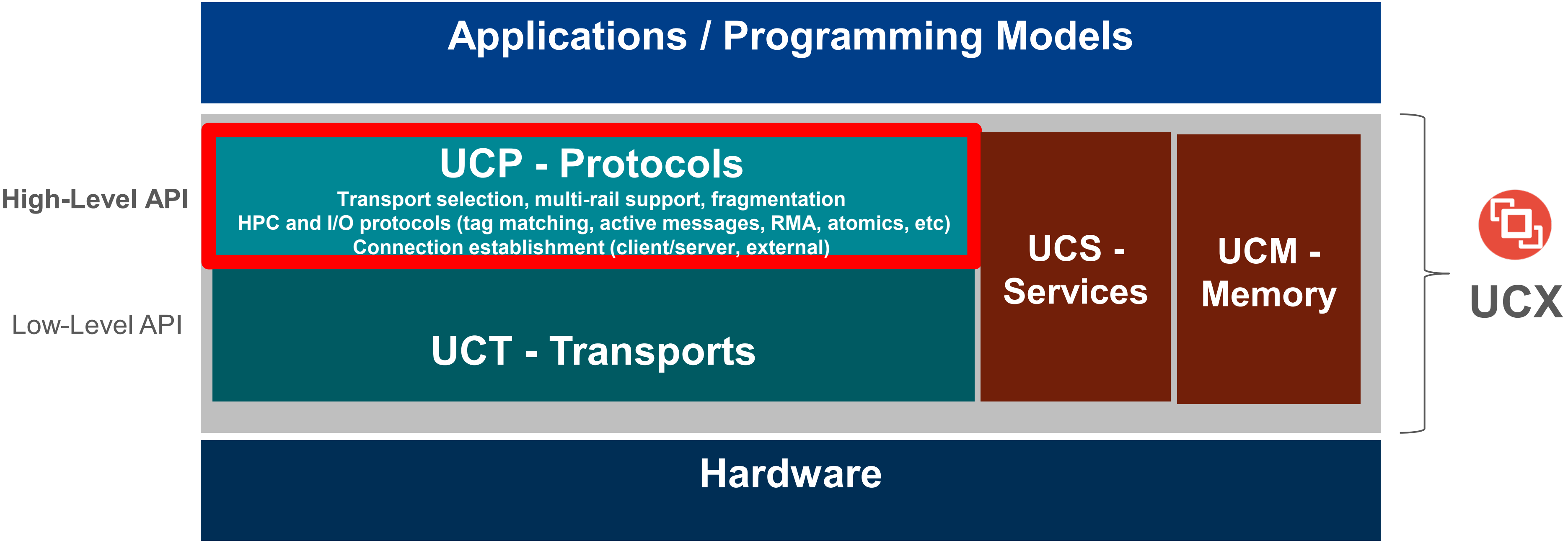
Hardware

Clarifications

- UCX is not a device driver
- UCX is a communication framework
  - Close-to-hardware API layer
  - Providing an access to hardware’s capabilities
- UCX relies on drivers supplied by vendors

API Overview

UCP – Protocol Layer



UCP Protocol

"Simplicity is the ultimate sophistication."

Leonardo da Vinci

Protocol Layer

- Selects the best network for the application
  - Does not have to be the same vendor
- Optimized by default
  - Protocols are optimized for the message size and underlying network semantics
  - Intelligent fragmentation
- Multi-rail, multi-interconnect communication
- Emulates unsupported semantics in software
  - No “ifdefs” in user code
  - Software atomics, tag-matching, etc.
- Abstracts connection setup
- Handles 99% of “corner” cases
  - Network out of resources
  - Reliability
  - No message size limit
  - ….and many more

UCP Objects

ucp_context_h

- A global context for the application; holds the memory registrations and global device context. For example, a hybrid MPI/SHMEM library may create one context for MPI, and another for SHMEM.

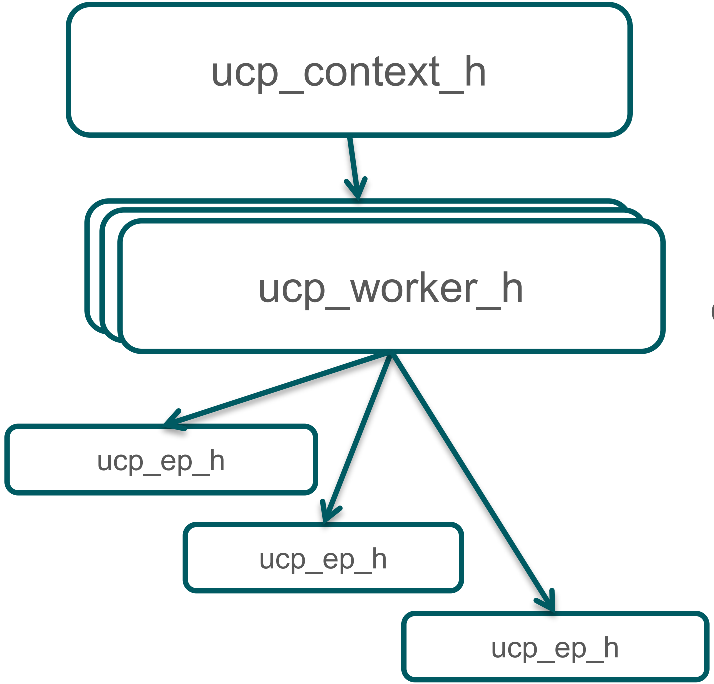
ucp_worker_h

- Communication and progress engine context; handles transport related resources: handling interrupts, connection establishment, completion events, etc. One possible usage is to create one worker per thread.

ucp_ep_h

- Represents a connection from a local worker to a remote worker. All send operations are performed on an end point.

A diagram on how different UCP Objects are related to each other



One instance per application process

Can have multiple workers to progress communication

Can have multiple end-points for communication peers

UCP Objects (Cont…)

ucp_mem_h

- A handle to an allocated or registered memory in the local process. Contains details describing the memory, such as address, length etc.

ucp_rkey_h

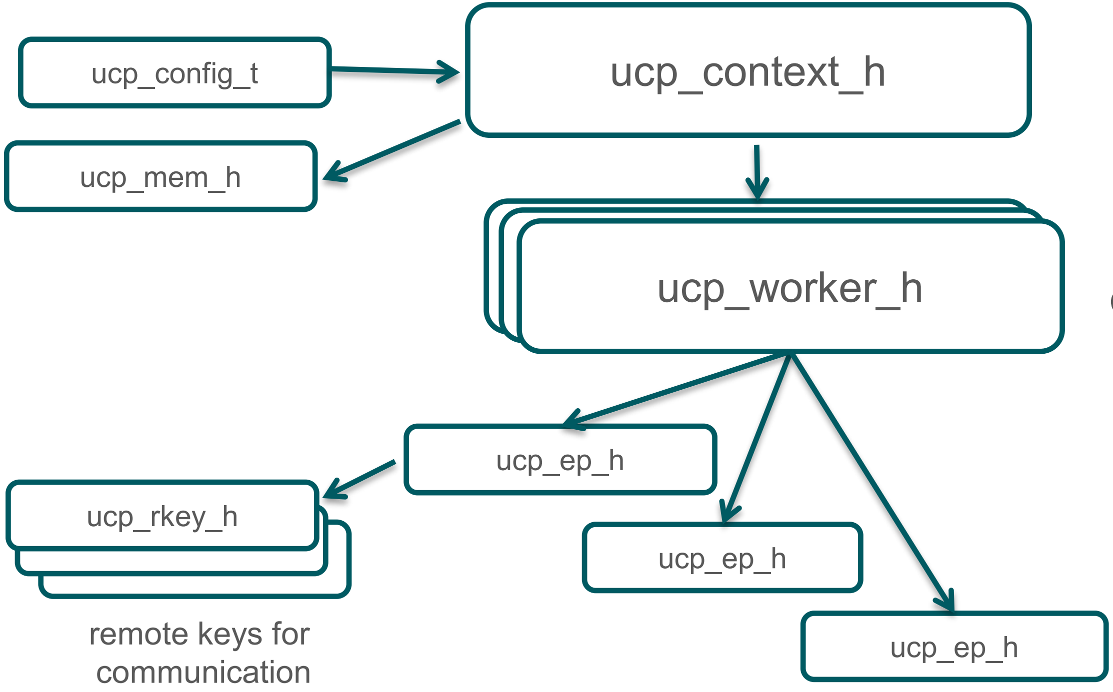
- Remote key handle, communicated to remote peers to enable access to the memory region. Contains an array of uct_rkey_t's.

ucp_config_t

- Configuration for ucp_context_h. Loaded from the run-time to set environment parameters for UCX.

© 2021 UCF Consortium

A diagram on how different UCP Objects are related to each other



Once instance per application process

Can have multiple workers to progress communication

Can have multiple end-points for communication peers

UCP Initialization

- ucp_init (const ucp_params_t ∗ params, const ucp_config_t ∗ config, ucp_context_h ∗ context_p)

ucp_context_h

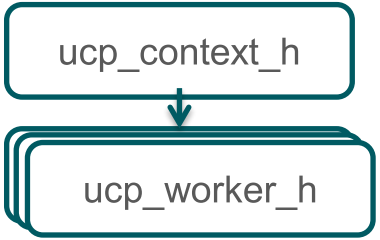
in	*config*	UCP configuration descriptor allocated through ucp_config_read() routine.
in	*params*	User defined tunings for the UCP application context.
out	*context_p*	Initialized UCP application context.

- This routine creates and initializes a UCP application context.
- This routine checks API version compatibility, then discovers the available network interfaces, and initializes the network resources required for discovering of the network and memory related devices. This routine is responsible for initialization all information required for a particular application scope, for example, MPI application, OpenSHMEM application, etc.

- Related routines: ucp_cleanup, ucp_get_version

UCP Initialization

- ucs_status_t ucp_worker_create (ucp_context_h context, ucs_thread_mode_t thread_mode, ucp_worker_h *worker_p)



in	*context*	Handle to UCP application context.
in	*thread_mode*	Thread safety mode for the worker object and resources associated with it.
out	*worker_p*	A pointer to the worker object allocated by the UCP library

- This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. At the same time, an application context can create multiple workers in order to enable concurrent access to communication resources. For example, application can allocate a dedicated worker for each application thread, where every worker can be progressed independently of others.

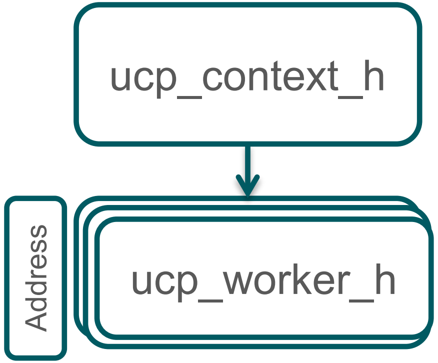
- Related routines: ucp_worker_destroy, ucp_worker_get_address, ucp_worker_release_address, ucp_worker_progress, ucp_worker_fence, ucp_worker_flush

UCP Initialization

- ucs_status_t ucp_worker_get_address (ucp_worker_h worker, ucp_address_t ** address_p, size_t * address_length_p)

in	*worker*	Worker object whose address to return.
out	*address_p*	A pointer to the worker address.
out	*address_length_p*	The size in bytes of the address.

- This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library in order to connect to this worker. The memory for the address handle is allocated by this function and must be released by using ucp_worker_release_address() routine.



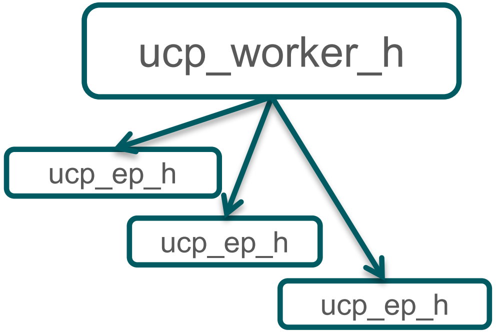
UCP Initialization

- ucs_status_t ucp_ep_create (ucp_worker_h worker, const ucp_address_t ∗ address, ucp_ep_h ∗ ep_p)

in	*worker*	Handle to the worker; the endpoint is associated with the worker.
in	*address*	Destination address; the address must be obtained using ucp_worker_get_↩ address() routine.
out	*ep_p*	A handle to the created endpoint.

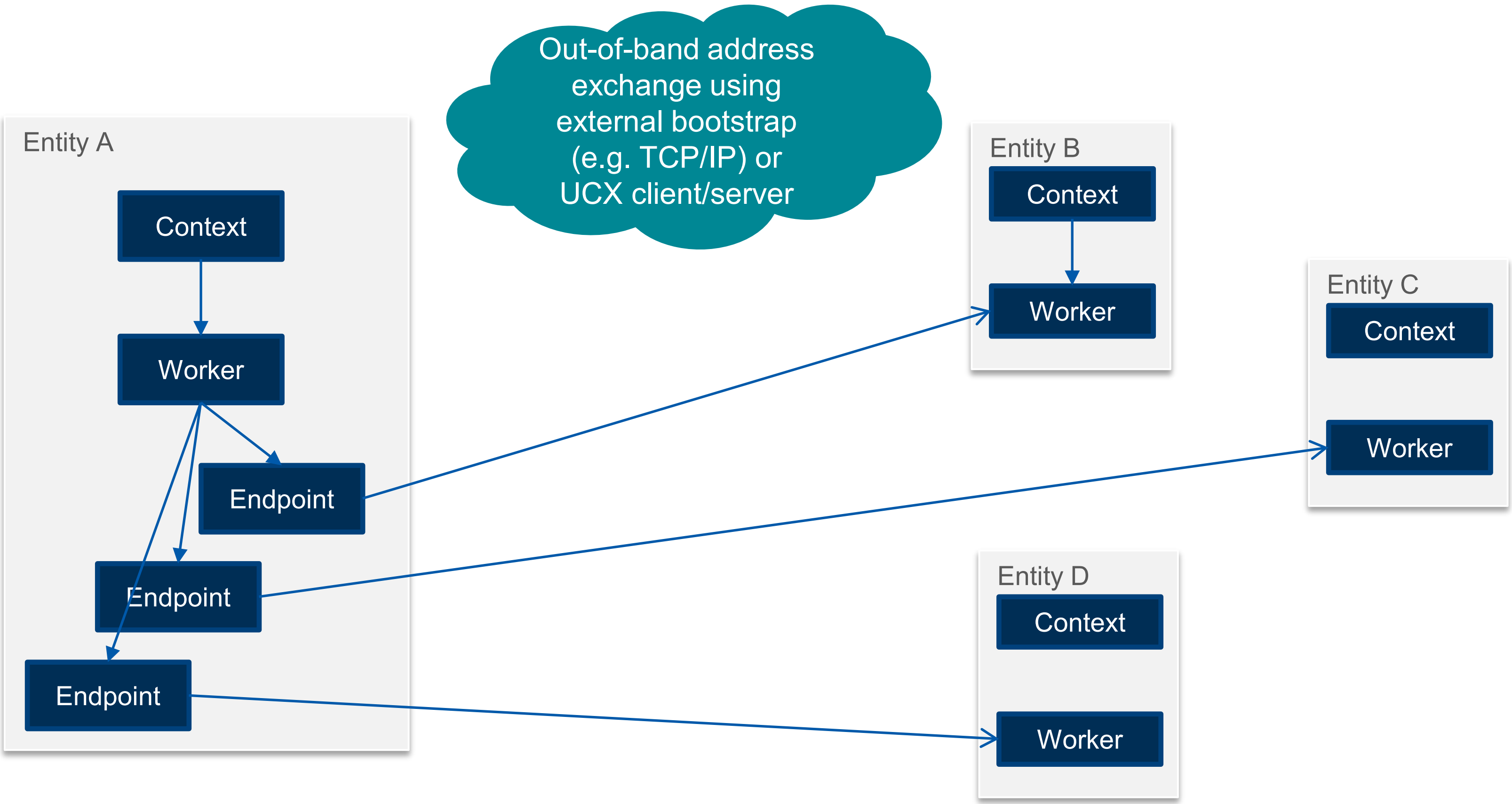
- This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker. This function is non-blocking, and communications may begin immediately after it returns. If the connection process is not completed, communications may be delayed. The created endpoint is associated with one and only one worker.

- Related routines: ucp_ep_flush, ucp_ep_fence, ucp_ep_destroy



© 2021 UCF Consortium

UCP API


Out-of-band address exchange using external bootstrap (e.g. TCP/IP) or UCX client/server
Entity A
Context
Worker
Endpoint
Endpoint
Endpoint
Entity B
Context
Worker
Entity C
Context
Worker
Entity D
Context
Worker

Putting it together: Implementing "Hello World" using UCP

Application Flow

Initialization App. Context	Bootstrap (TCP/IP)	Setup Workers/ Endpoints	Start Communication	Disconnect & Cleanup
Environment Parameters **Create Application Context** (ucp_context_h)	Establish socket Connection (client/server) **Create Local Worker Obj** (ucp_worker_h) Send Worker Obj Address to peer over socket(client)	**Create local Endpoint** **Create Remote Endpoint**	**Send/Recv Tag or RDMA (e.g. put/get)** **Progress Communication** **print "Hello World"**	**Disconnect Endpoint Clean up**

TUTORIAL BREAK

Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview,	Jeff		
	worker and endpoint creation			
10:00 - 10:10	**BREAK**			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	**BREAK**			
11:10 - 11:50	UCX Advanced Topics - bindings,	Matt, Yossi, Oscar		
	OpenMPI and OpenSHMEM integration			

We will return at ***10:10 US PDT***

Demo of Hello World

- Code walk-through

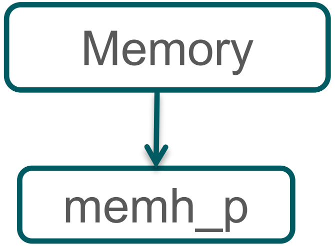
Putting it together: Implementing “Hello World” using UCP – Q&A

Application Flow

Initialization App. Context	Bootstrap (TCP/IP)	Setup Workers/ Endpoints	Start Communication	Disconnect & Cleanup
Environment Parameters **Create Application Context** (ucp_context_h)	Establish socket Connection (client/server) **Create Local Worker Obj** (ucp_worker_h) Send Worker Obj Address to peer over socket(client)	**Create local Endpoint** **Create Remote Endpoint**	**Send/Recv Tag or RDMA (e.g. put/get)** **Progress Communication** **print “Hello World”**	**Disconnect Endpoint Clean up**

UCP Memory Management

- ucs_status_t ucp_mem_map (ucp_context_h context, void **address_p, size_t length, unsigned flags, ucp_mem_h *memh_p)

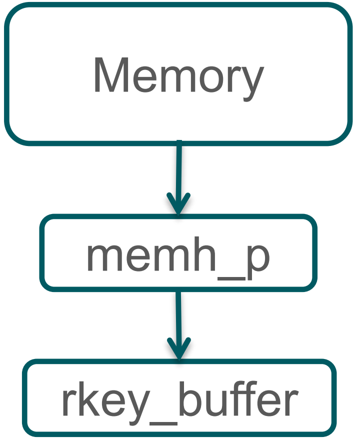


in	*context*	Application context to map (register) and allocate the memory on.
in,out	*address_p*	If the pointer to the address is not NULL, the routine maps (registers) the memory segment. if the pointer is NULL, the library allocates mapped (registered) memory segment and returns its address in this argument.
in	*length*	Length (in bytes) to allocate or map (register).
in	*flags*	Allocation flags (currently reserved - set to 0).
out	*memh_p*	UCP handle for the allocated segment.

- This routine maps or/and allocates a user-specified memory segment with UCP application context and the network resources associated with it.
- Related routines: ucp_mem_unmap

UCP Memory Management

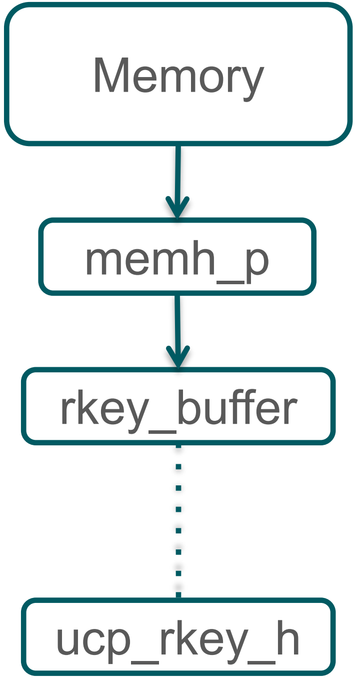
- ucs_status_t ucp_rkey_pack (ucp_context_h context, ucp_mem_h memh, void ∗∗rkey_buffer_p, size_t ∗size_p)



in	*context*	Application context which was used to allocate/map the memory.
in	*memh*	Handle to memory region.
out	*rkey_buffer_p*	Memory buffer allocated by the library. The buffer contains packed RKEY.
out	*size_p*	Size (in bytes) of the packed RKEY.

- This routine allocates memory buffer and packs into the buffer a remote access key (RKEY) object. RKEY is an opaque object that provides the information that is necessary for remote memory access. This routine packs the RKEY object in a portable format such that the object can be unpacked on any platform supported by the UCP library.
- Related routines: ucp_rkey_buffer_release

UCP Memory Management



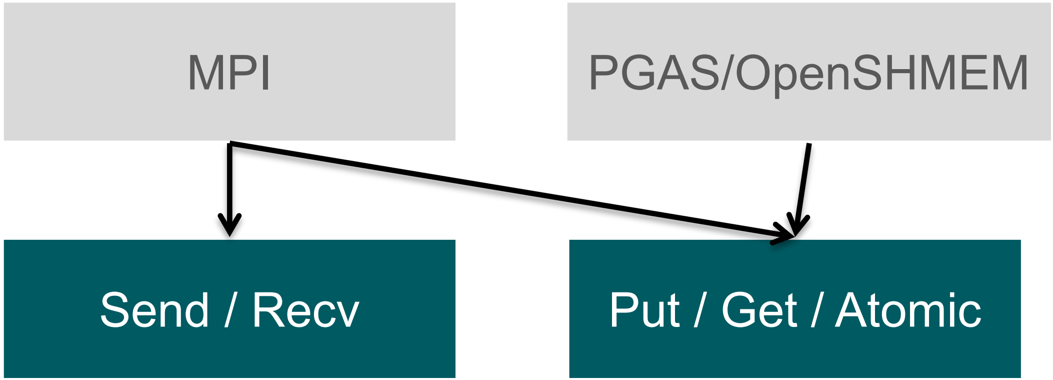
- ucs_status_t ucp_ep_rkey_unpack (ucp_ep_h ep, void ∗rkey_buffer, ucp_rkey_h ∗rkey_p)

in	*ep*	Endpoint to access using the remote key.
in	*rkey_buffer*	Packed rkey.
out	*rkey_p*	Remote key handle.

- This routine unpacks the remote key (RKEY) object into the local memory such that it can be accesses and used by UCP routines. The RKEY object must be packed using the ucp_rkey_pack() routine. Application code should not make any alterations to the content of the RKEY buffer.

- Related routines: ucp_rkey_destroy

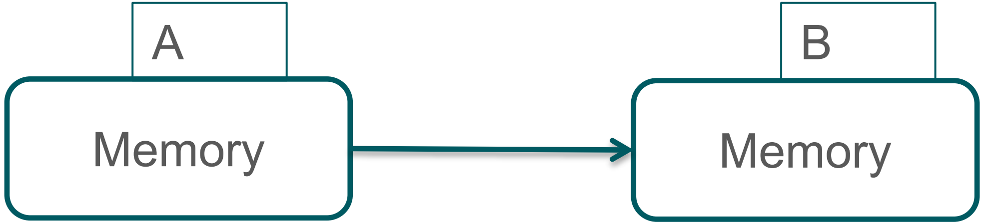
Communication Directives



Put operation

- ucs_status_ptr_t ucp_put_nbx (ucp_ep_h ep, const void ∗ buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t ∗ param)

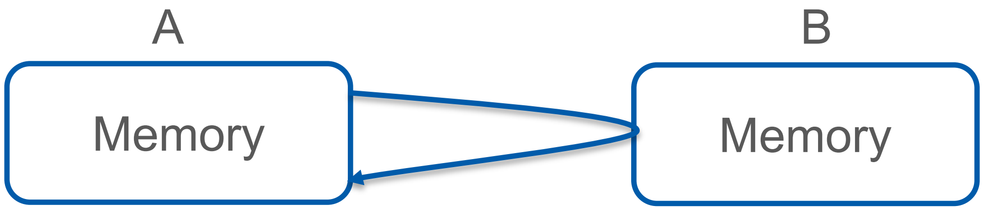
in	*ep*	Remote endpoint handle.
in	*buffer*	Pointer to the local source address.
in	*count*	Number of elements of type ucp_request_param_t::datatype to put. If ucp_request_param_t::datatype is not specified, the type defaults to ucp_dt_make_contig(1), which corresponds to byte elements.
in	*remote_addr*	Pointer to the destination remote memory address to write to.
in	*rkey*	Remote memory key associated with the remote memory address.
in	*param*	Operation parameters, see ucp_request_param_t



Get operation

- ucs_status_ptr_t ucp_get_nbx (ucp_ep_h ep, void ∗ buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t ∗ param)

in	*ep*	Remote endpoint handle.
in	*buffer*	Pointer to the local destination address.
in	*count*	Number of elements of type ucp_request_param_t::datatype to put. If ucp_request_param_t::datatype is not specified, the type defaults to ucp_dt_make_contig(1), which corresponds to byte elements.
in	*remote_addr*	Pointer to the source remote memory address to read from.
in	*rkey*	Remote memory key associated with the remote memory address.
in	*param*	Operation parameters, see ucp_request_param_t.



© 2021 UCF Consortium

Send

- ucs_status_ptr_t ucp_tag_send_nbx (ucp_ep_h ep, const void ∗ buffer, size_t count, ucp_tag_t tag, const ucp_request_param_t ∗ param)

in	*ep*	Destination endpoint handle.
in	*buffer*	Pointer to the message buffer (payload).
in	*count*	Number of elements to send
in	*tag*	Message tag.
in	*param*	Operation parameters, see ucp_request_param_t

© 2021 UCF Consortium

Receive

- ucs_status_ptr_t ucp_tag_recv_nbx (ucp_worker_h worker, void ∗ buffer, size_t count, ucp_tag_t tag, ucp_tag_t tag_mask, const ucp_request_param_t ∗ param)

in	*worker*	UCP worker that is used for the receive operation.
in	*buffer*	Pointer to the buffer to receive the data to.
in	*count*	Number of elements to receive
in	*tag*	Message tag to expect.
in	*tag_mask*	Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.
in	*param*	Operation parameters, see ucp_request_param_t

ucs_status_ptr_t

- UCS_OK - The send operation was completed immediately.
- UCS_PTR_IS_ERR(_ptr) - The send operation failed.
- otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message. The application is responsible to release the handle using ucp_request_release() routine.
- Request handling
  - int ucp_request_is_completed (void * request)
  - void ucp_request_release (void * request)
  - void ucp_request_cancel (ucp_worker_h worker, void * request)

Send-Sync

- ucs_status_ptr_t ucp_tag_send_sync_nbx (ucp_ep_h ep, const void ∗ buffer, size_t count, ucp_tag_t tag, const ucp_request_param_t ∗ param)

in	*ep*	Destination endpoint handle.
in	*buffer*	Pointer to the message buffer (payload).
in	*count*	Number of elements to send
in	*tag*	Message tag.
in	*param*	Operation parameters, see ucp_request_param_t

Atomic operations

- ucs_status_ptr_t ucp_atomic_op_nbx(ucp_ep_h ep, ucp_atomic_op_t opcode, const void *buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t *param)

in	*ep*	UCP endpoint.
in	*opcode*	One of ucp_atomic_op_t.
in	*buffer*	Address of operand for the atomic operation. See 6.142 for exact usage by different atomic operations.
in	*count*	Number of elements in *buffer* and *result*. The size of each element is specified by ucp_request_param_t::datatype
in	*remote_addr*	Remote address to operate on.
in	*rkey*	Remote key handle for the remote memory address.
in	*param*	Operation parameters, see ucp_request_param_t.

Example: Implementing "Hello World" using UCP

Communication Flow

Initialization App. Context	Bootstrap (TCP/IP)	Setup Workers/ Endpoints	Start Communication	Disconnect & Cleanup
Environment Parameters **Create Application Context** (ucp_context_h)	Establish socket Connection (client/server) **Create Local Worker Obj** (ucp_worker_h) Send Worker Obj Address to peer over socket(client)	**Create local Endpoint** **Create Remote Endpoint**	**Send/Recv Tag or RDMA (e.g. put/get)** **Progress Communication** **print "Hello World"**	**Disconnect Endpoint Clean up**

UCP Example/Demo Revisited

- "Hello World with UCP"
  - Using two nodes with IB
  - UCX for communication - send/recv
  - Finalization

Example: Implementing “Hello World” using UCP – Q&A

Communication Flow

Initialization App. Context	Bootstrap (TCP/IP)	Setup Workers/ Endpoints	Start Communication	Disconnect & Cleanup
Environment Parameters **Create Application Context** (ucp_context_h)	Establish socket Connection (client/server) **Create Local Worker Obj** (ucp_worker_h) Send Worker Obj Address to peer over socket(client)	**Create local Endpoint** **Create Remote Endpoint**	**Send/Recv Tag** **or** **RDMA (e.g. put/get)** **Progress Communication** **print** **“Hello World”**	**Disconnect Endpoint** **Clean up**

Support for Accelerators (e.g. GPUs)

- UCX was designed from the beginning with accelerators in mind
- Accelerators are structured as transports
  - The accelerator transport abstractions is responsible to transferring data between the host and accelerator memory
  - Protocols are agnostic to the types of accelerators
  - No need to change higher level logic in order to add new accelerators
- Communicating accelerator memory works the same as passing host memory
  - Passing a pointer to the accelerator memory in UCP API works
  - The user can pass runtime flags to override internal memory detection
- The programming model doesn't need to deal with accelerator logic/code

UCP Accelerator Example/Demo

- "Hello World with UCP using GPUs"
  - Show how to use UCP to communicate between CPU and GPUs

TUTORIAL BREAK

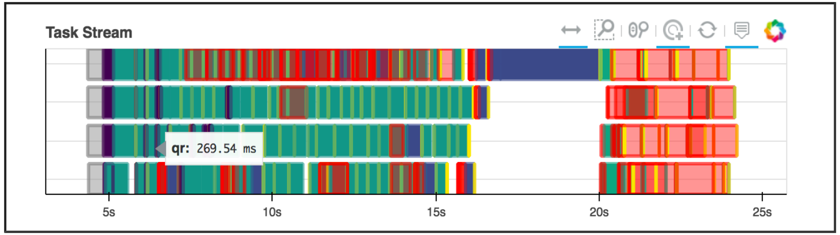
Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview,	Jeff		
	worker and endpoint creation			
10:00 - 10:10	BREAK			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	BREAK			
11:10 - 11:50	UCX Advanced Topics - bindings,	Matt, Yossi, Oscar		
	OpenMPI and OpenSHMEM integration			

We will return at ***11:15 US PDT***

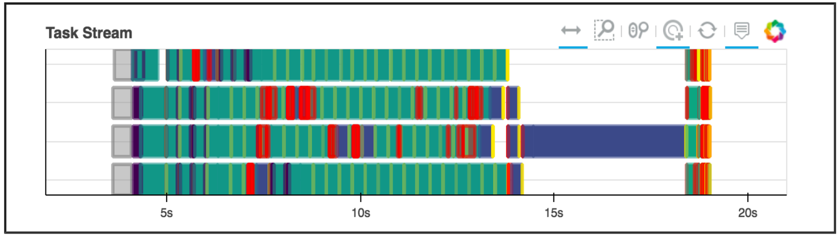
Other bindings: UCX-py, UCX-java, UCX-go

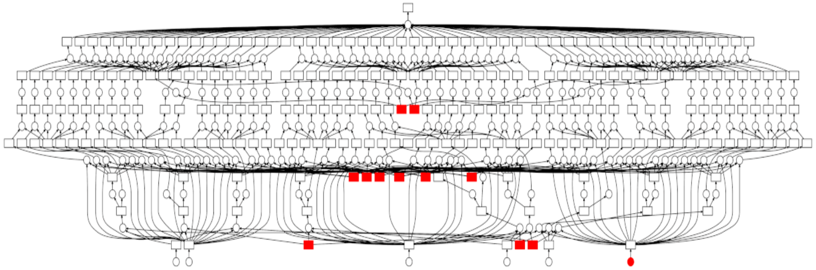
Helping converge HPC to Data Sciences, AI, and BigData

Before UCX:



After UCX:





Science loves Python

- Easy to learn, free; numpy is similar to MATLAB

RAPIDS

- Open-source data science python libraries
- Single thread multi accelerator

DASK

- Distributed tasking framework for python

UCX

- Client-server model, python bindings, efficient RDMA, etc.

UCX-py: https://github.com/rapidsai/ucx-py
UCX-java, UCX-go: https://github.com/openucx/ucx
Source: *Matthew Rocklin, Rick Zamora,* **Experiments in High Performance Networking with UCX and DGX**
https://blog.dask.org/2019/06/09/ucx-dgx

Example of UCX-py (via RAPIDS)

```python
async def test_send_recv_numpy(size, dtype, blocking_progress_mode):
    ucp.init(blocking_progress_mode=blocking_progress_mode)

    msg = np.arange(size, dtype=dtype)
    msg_size = np.array([msg.nbytes], dtype=np.uint64)

    listener = ucp.create_listener(
        make_echo_server(lambda n: np.empty(n, dtype=np.uint8))
    )
    client = await ucp.create_endpoint(ucp.get_address(), listener.port)
    await client.send(msg_size)
    await client.send(msg)
    resp = np.empty_like(msg)
    await client.recv(resp)
    np.testing.assert_array_equal(resp, msg)
```

Example extracted from https://github.com/rapidsai/ucx-py/blob/branch-0.22/tests/test_send_recv.py

AGENDA UPDATE

Time	Topic	Presenters	Slides	Notes
9:00 - 9:20	UCX Tutorial and Ecosystem Introduction	Gilad		
9:20 - 10:00	UCX Basics - networking overview,	Jeff		
	worker and endpoint creation			
10:00 - 10:10	BREAK			
10:10 - 10:20	Hello World part 1	Matt, Jeff		
10:20 - 10:35	UCX memory management	Jeff, Oscar		
10:35 - 10:45	Hello World Demo, part 2	Matt, Jeff		
10:45 - 11:00	GPU discussion/demo	Oscar, Matt		
11:00 - 11:10	BREAK			
11:10 - 11:50	UCX Advanced Topics - bindings,	Matt, Yossi, Oscar		
	OpenMPI and OpenSHMEM integration			

OPEN MPI INTEGRATION WITH UCX

Overview

- Open MPI (OMPI) supports UCX starting OMPI v1.10.
- UCX is a PML component in OMPI.
  To enable UCX:
  - mpirun -mca pml ucx … <APP>
- OMPI is integrated with the UCP layer.

Overview

- UCX mca parameters:
  – pml_ucx_verbose, pml_ucx_priority

- UCX environment parameters:
  – ucx_info -f

- For example:
  mpirun -mca pml ucx -x UCX_NET_DEVICES=mlx5_0:1 … <APP>

Overview

- UCX will select the best transport to use.
- Supported transports include
  - IB - ud, rc, dc, accelerated verbs
  - shared memory
  - uGNI
- OMPI uses Full/Direct modes with UCX.
- UCX will connect the ranks.

UCP Main Objects - Recap

- **ucp_context_h**

A global context for the application - a single UCP communication instance.
Includes communication resources, memory and other communication information directly associated with a specific UCP instance.

- **ucp_worker_h**

Represents an instance of a local communication resource and an independent progress of communication. It contains the uct_iface_h's of all selected transports.
One possible usage is to create one worker per thread.

- **ucp_ep_h**

Represents a connection to a remote worker.
It contains the uct_ep_h's of the active transports.

UCX Main Objects - Recap

- **ucp_mem_h**

A handle to an allocated or registered memory in the local process. Contains details describing the memory, such as address, length etc.

- **ucp_rkey_h**

Remote key handle, communicated to remote peers to enable an access to the memory region. Contains an array of uct_rkey_t's.

- **ucp_config_t**

Configuration for ucp_context_h. Loaded from the run-time to set environment parameters for UCX.

OMPI - UCX Stack

APPLICATION

OMPI

PML UCX

UCX LIBRARY

DRIVER

HARDWARE

© 2021 UCF Consortium

Init Stage

MPI_Init → ompi_mpi_init

mca_pml_ucx_open

} OpenMPI

ucp_config_read

ucp_init

ucp_config_release

} UCX

Init Stage - Cont

mca_pml_ucx_init

ucp_worker_create

ucp_worker_get_address

ucp_worker_release_address

Init Stage - Cont

opal_progress_register(mca_pml_ucx_progress)

↓

ucp_worker_progress

Send Flow

MPI_Isend → mca_pml_ucx_isend

mca_pml_ucx_add_proc

ucp_ep_create

ucp_tag_send_nb → uct_ep_am_short, uct_ep_am_bcopy, uct_ep_am_zcopy

Send Flow - Cont

- If the send request isn't completed
  → progress it.

- Once completed
  → callback function is invoked

- Contiguous and non-contiguous datatypes are supported.

Receive Flow

MPI_Irecv → mca_pml_ucx_irecv

ucp_tag_recv_nb

- Eager Receive
- Rendezvous

Receive Flow - Cont

- Expected & Unexpected queues are used
- Can probe with ucp_tag_probe_nb
  → ucp_tag_msg_recv_nb
- If the receive request isn’t completed
  → progress it.
- Once completed
  → callback function is invoked

Progress Flow

opal_progress

mca_pml_ucx_progress

ucp_worker_progress → uct_worker_progress

* Send/Receive Finished:

mca_pml_ucx_send/recv_completion

Finalization Stage

MPI_Finalize → ompi_mpi_finalize

mca_pml_ucx_del_procs

* Per remote peer:

ucp_ep_destroy

Finalization Stage - Cont

mca_pml_ucx_cleanup

opal_progress_unregister (mca_pml_ucx_progress)

ucp_worker _destroy

mca_pml_ucx_close

ucp_cleanup

OPENSHMEM INTEGRATION WITH UCX

OpenSHMEM - Overview

- PGAS Library
- One-sided Communication
- Atomic operations
- Collectives

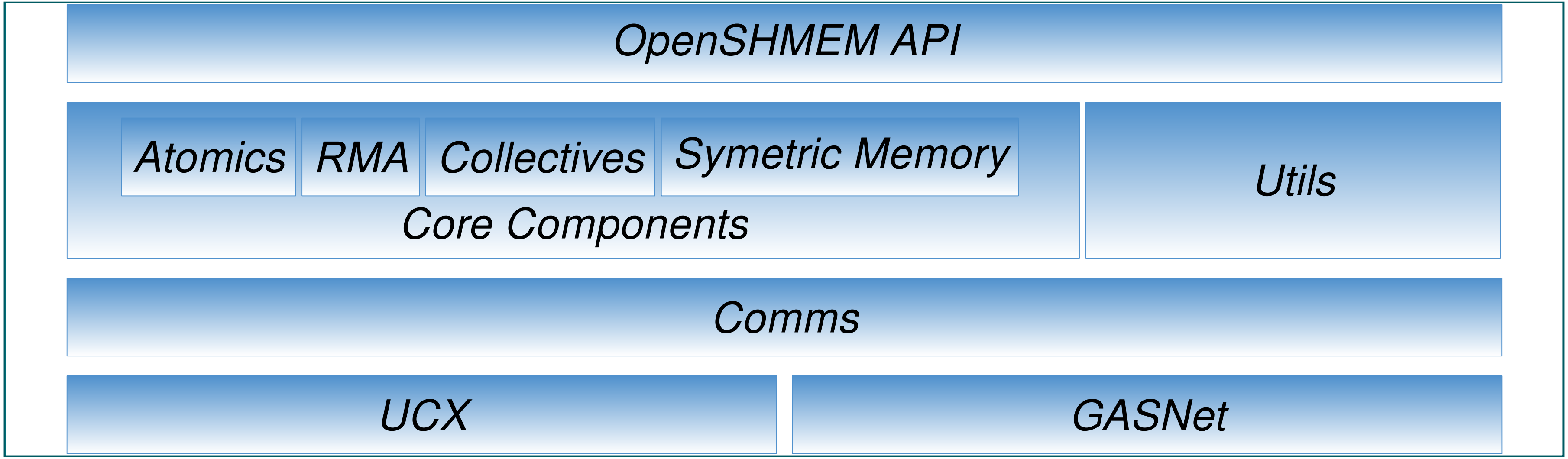
Symmetric Memory

- All Processing Elements (PE's) share an address space (symmetric heap)
- Symmetric heap is allocated on startup
- Heapsize can be customized via environment variable SMA_SYMMETRIC_SIZE
- Symmetric data objects must be allocated with shmem_malloc
- Symmetric data objects are accessible by remote PEs

Shared global address space

- Global and static variables are symmetric objects
- Accessible by remote PEs

OpenSHMEM Reference Implementation



http://www.openshmem.org/

Initialization

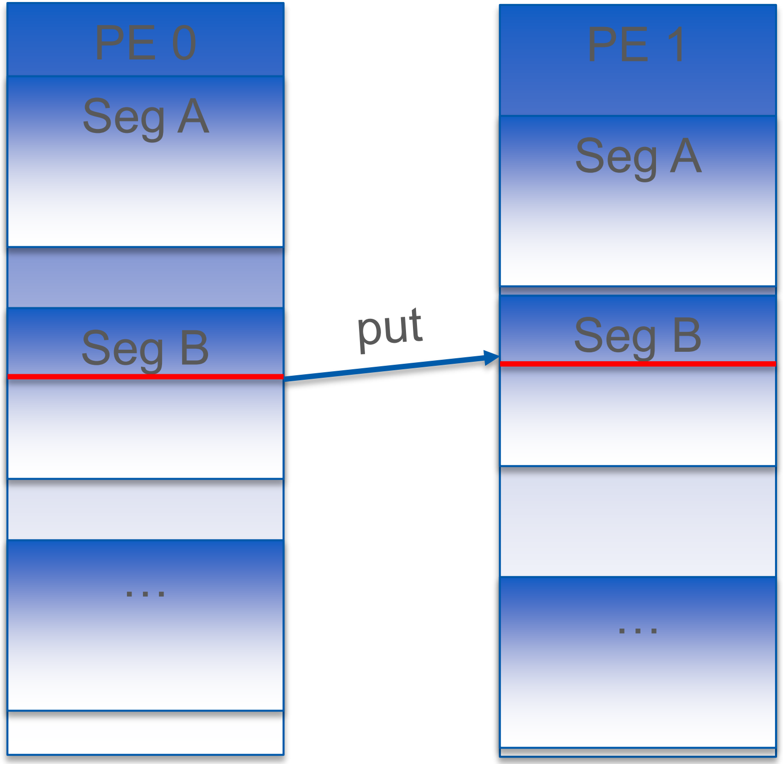
- shmem_init
  - shmemi_comms_init
    - ...
    - ucp_config_read
    - ucp_init
    - ucp_config_release
    - ucp_worker_create
    - init_memory_regions (more on this later)
    - ...
    - ucp_ep_create (for each PE)
    - ucp_config_release
    - ...

Memory registration

- OpenSHMEM registers global data (data and bss segment) and the symmetric heap
- ucp_mem_map maps the memory with the ucp context (returning ucp_mem_h)

Translating symmetric addresses

- For RMA operations UCP needs Remote Memory Handle (remote key or rkey)
- To access a remote address the rkey is needed
- Look up rkey with ***find_seg***
- Translate local buffer address into remote buffer address



RMA put / get / atomics

- shmem_<TYPENAME>_<put, get, atomic_op>

  - find_seg
  - translate_symmetric_address

  - depending on the API pick:
    - ucp_put
    - ucp_get
    - ucp_atomic_<op,size>

UCP picks the right internal UCT calls depending on the message size, and operation (e.g. atomics)

Compiling UCX for OpenMPI and OpenSHMEM

UCX Build

```
$ ./autogen.sh
$ ./contrib/configure-release --prefix=$PWD/install
$ make -j && make install
```

OpenMPI UCX Build

```
<openmpi_root>$ ./autogen.pl
$  --prefix=$PWD/install \
--with-ucx=$PWD/ucx
$ make -j && make install
```

OpenSHMEM UCX Build

```
<openshmem_root>$ ./autogen.pl
$ ./contrib/configure-release --with-comms-layer=ucx \
--with-ucx-root=$PWD/install \
--with-rte-root=$PWD/install \
--prefix=$PWD/install
$ make -j && make install
```

Summary

- We covered an overview of UCX with a focus on UCP
  - Demonstrated simple examples
- UCX has been integrated with multiple programming models and frameworks:
  - MPI: Open MPI, MPICH
  - OpenSHMEM: Reference Implementation, OSHMEM
  - Dask, Spark, BlazingSQL, etc
- Supports multiple transports and accelerators
  - IB/RoCE: RC, UD, DCT, CM
  - Aries,Slingshot: FMA, SMSG, BTE
  - Shared Memory: SysV, Posix, CMA, KNEM, XPMEM
  - Accelerators: CUDA, ROCm
- Bindings and GPU integration allow for focus on user-level programming rather than on low-level communication APIs
  - UCX-Py and other bindings provide paths to use UCX for HPC apps
  - GPU integration helps to accelerate DGX cluster communication

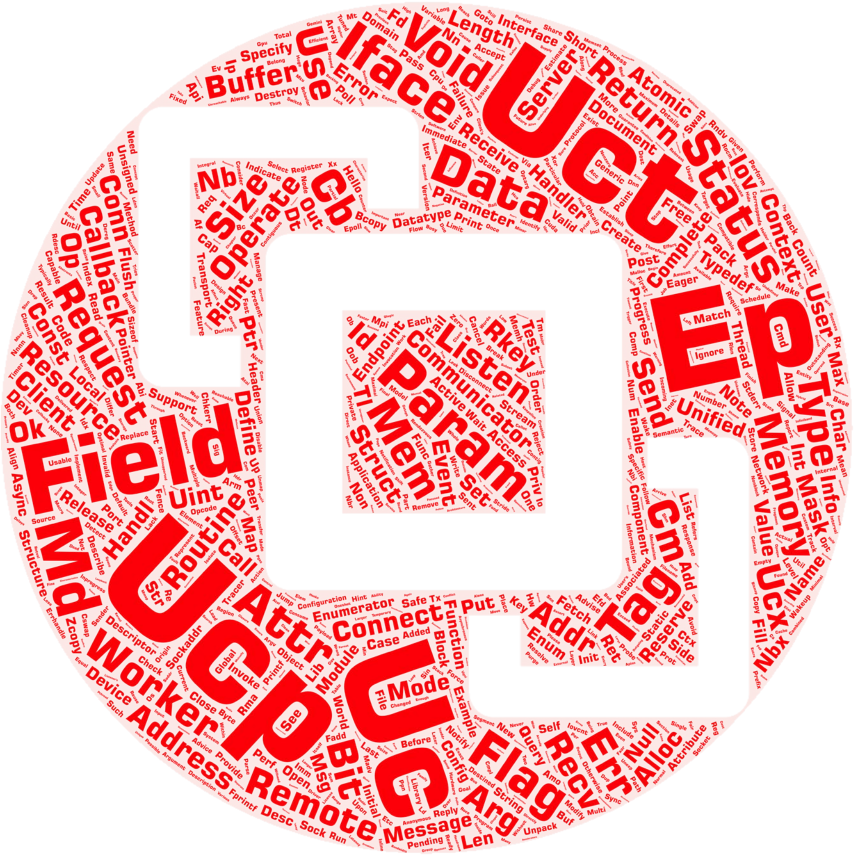
© 2021 UCF Consortium

Unified Communication - X
Framework

Web:
www.openucx.org
https://github.com/openucx/ucx

Mailing List:
https://elist.ornl.gov/mailman/listinfo/ucx-group
ucx-group@elist.ornl.gov

Q&A



Uct
Ucp
Uc
Ep
Field
Param
Status
Data
Iface
Void
Buffer
Memory
Request
Callback
Worker
Remote
Flag
Tag
Connect
Mode
Listen
Communicator

Acknowledgements

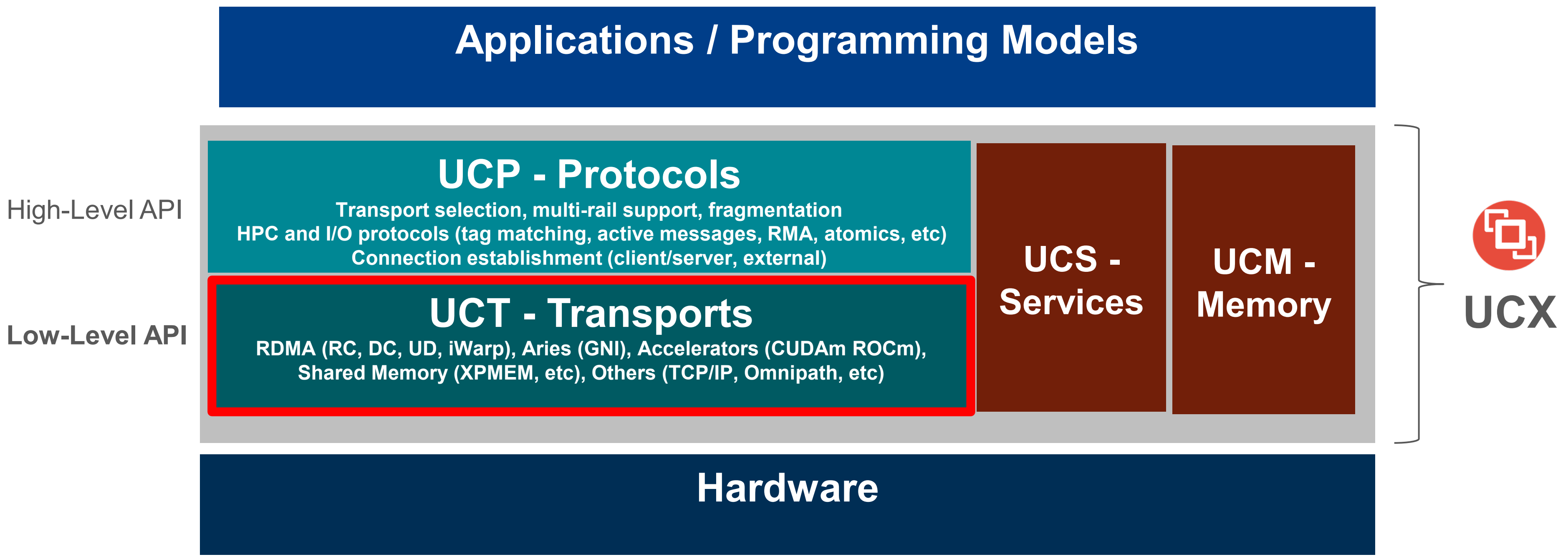
- This work was supported by the United States Department of Defense & used resources at Oak Ridge National Laboratory.
- This work was also partially supported by NSF CNS #2016701, "Rogues Gallery: A Community Research Infrastructure for Post-Moore Computing".

Backup Slides

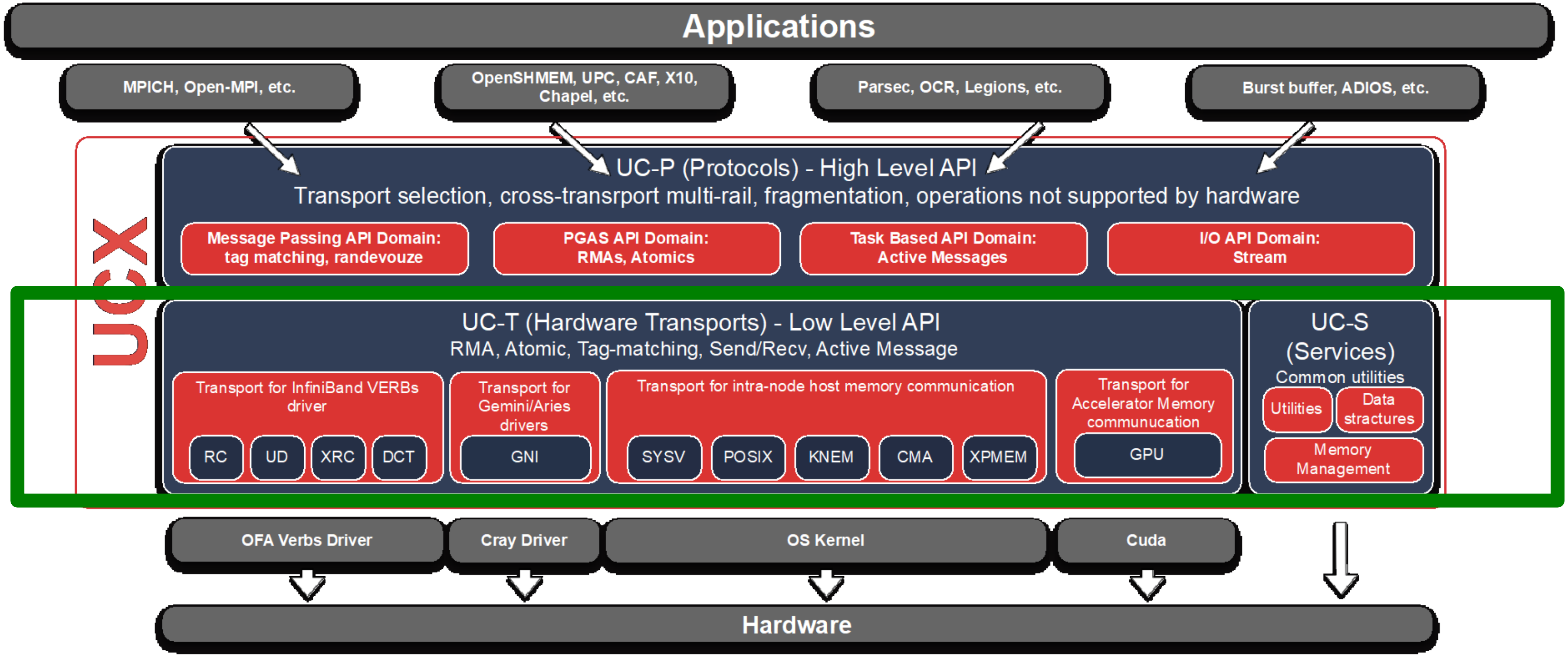
Note these slides were not presented at the tutorial and are provided for further reference only. They may not be completely up-to-date with the current UCX implementation.

UCT – Low-level API for Transports (Advanced Usage Only)

UCT – Transport Layer



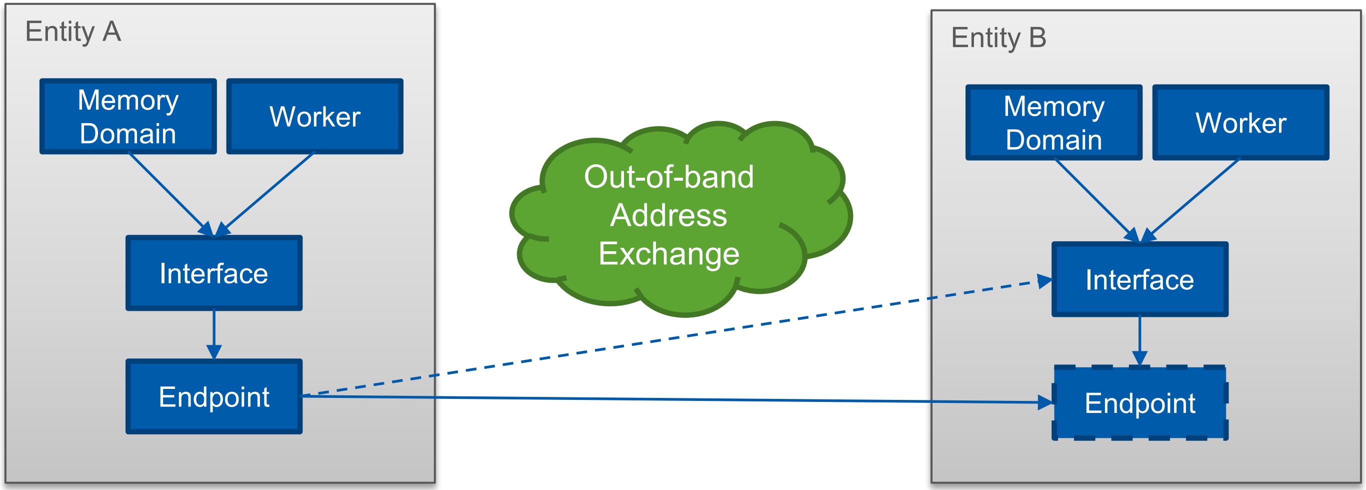
UCT – Transport Layer



UCT (Transport layer) objects

- uct_worker_h - A context for separate progress engine and communication resources. Can be either thread-dedicated or shared
- uct_md_h - Memory registration domain. Can register user buffers and/or allocate registered memory
- uct_iface_h - Communication interface, created on a specific memory domain and worker. Handles incoming active messages and spawns connections to remote interfaces
- uct_ep_h - Connection to a remote interface. Used to initiate communications

UCT initialization



Memory Domain Routines

- Register/de-register memory within the domain
  - Can potentially use a cache to speedup memory registration
- Allocate/de-allocate registered memory
- Pack memory region handle to a remote-key-buffer
  - Can be sent to another entity
- Unpack a remote-key-buffer into a remote-key
  - Can be used for remote memory access

UCT Communication Routines

- Not everything has to be supported
  - Interface reports the set of supported primitives
  - UCP uses this info to construct protocols
  - UCP implement emulation of unsupported directives
- Send active message (active message id)
- Put data to a remote memory (virtual address, remote key)
- Get data from a remote memory (virtual address, remote key)
- Perform an atomic operation on a remote memory:
  - Add
  - Fetch-and-add
  - Swap
  - Compare-and-swap
- Communication Fence and Flush (Quiet)

UCT Data Classes

- UCT communications have a size limit
  - Interface reports max. allowed size for every operation
  - Fragmentation, if required, should be handled by user / UCP
- Several data “classes” are supported
  - “short” – small buffer
  - “bcopy” – a user callback which generates data (in many cases, “memcpy” can be used as the callback)
  - “zcopy” – a buffer and it’s memory region handle. Usually large buffers are supported.
- Atomic operations use a 32 or 64 bit values

UCT Completion Semantics

- All operations are non-blocking
- Return value indicates the status:
  - OK – operation is completed
  - INPROGRESS – operation has started, but not completed yet
  - NO_RESOURCE – cannot initiate the operation right now. The user might want to put this on a pending queue, or retry in a tight loop
  - ERR_xx – other errors
- Operations which may return INPROGRESS (get/atomics/zcopy) can get a completion handle
  - User initializes the completion handle with a counter and a callback
  - Each completion decrements the counter by 1, when it reaches 0 – the callback is called

Memory Management and Data Path – Further Topics

Memory Allocation Strategies

- It is a limited resource
  - The goal is to maximize the availability of memory for the application
- Avoid O(n) memory allocations, where n is the number communication peers (endpoints)
- Keep the endpoint object as small as possible
- Keep the memory pools size limited
- Allocation has to be proportional to the number of in-flight-operations

Data Path

- Three main data paths:
  - Short messages – critical path
  - Medium messages
  - All the rest

Data Path / Short Messages

- Take care of the small-message case first
- Avoid function calls
- Avoid extra pointer dereference, especially store operations
- Avoid adding conditionals, if absolutely required use ucs_likely/ucs_unlikely macros
- Avoid bus-locked instructions (atomics)
- Avoid branches
- No malloc()/free() nor system calls
- Limit the scope of local variables (the time from first to last time it is used) - larger scopes causes spilling more variables to the stack
- Use benchmarks and performance analysis tools to analyze the impact on the latency and message rate

© 2021 UCF Consortium

Data Path / Medium Messages

- Avoid locks if possible. If needed, use spinlock, no mutex.
- Reduce function calls
- Move error and slow-path handling code to non-inline functions, so their local variables will not add overhead to the prologue and epilogue of the fast-path function.

Data Path / “Slow” Path

- Performance is still important
- No system calls / malloc() / free()
- It's ok to reasonable add pointer dereferences, conditionals, function calls, etc.
  - Having a readable code here is more important than saving one conditional or function call.
- Protocol-level performance considerations are more important here, such as fairness between connections, fast convergence, etc.
- Avoid O(n) complexity. As a thumb rule, all scheduling mechanisms have to be O(1).