OpenCPI - Open Component Portability Infrastructure

An Open Source Framework for Heterogeneous/Embedded Component-Based Systems and Applications

Introduction to OpenCPI

Introduction to OpenCPI

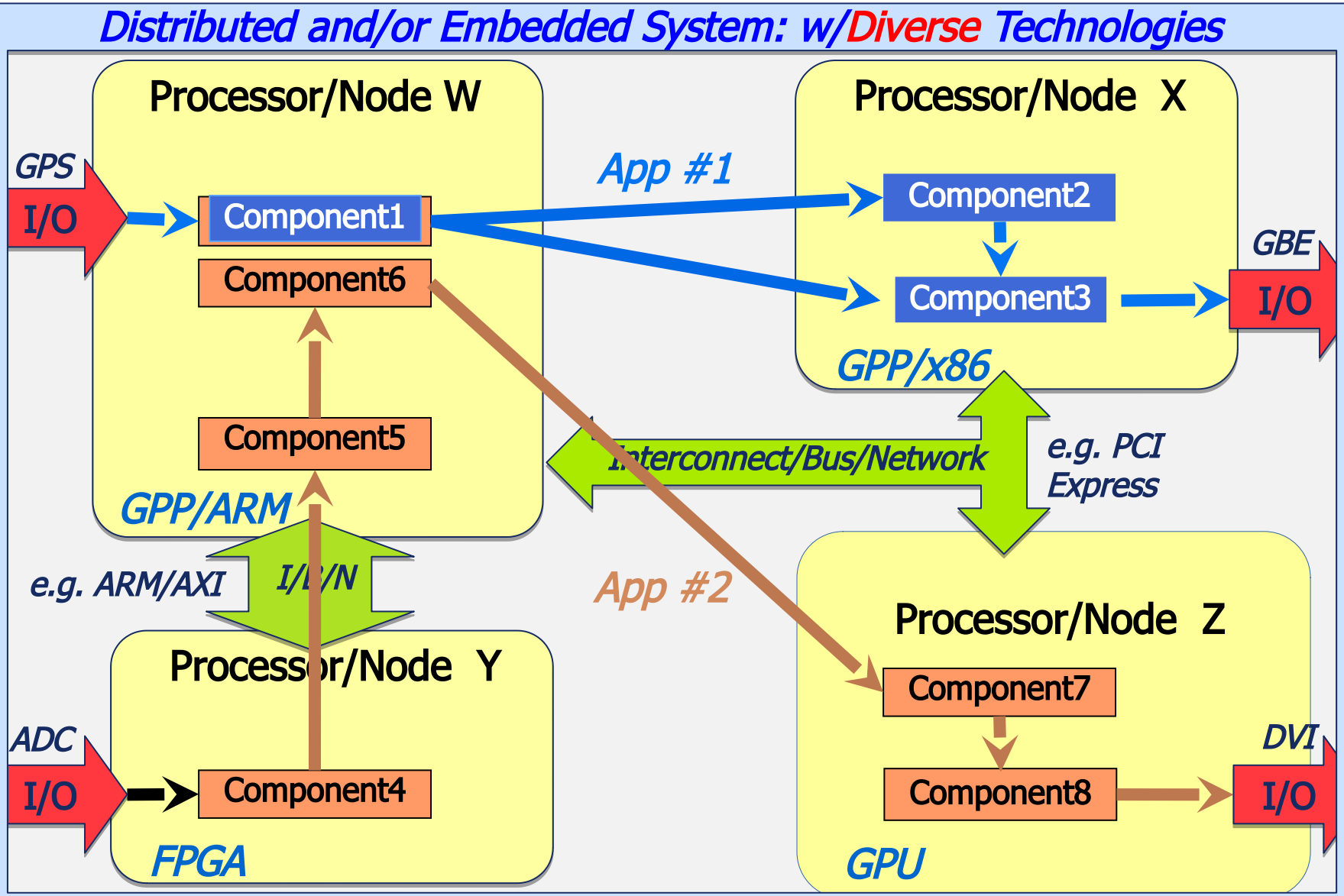
- Definition, Positioning, Priorities, Background
- Component Concepts and Terminology
- Application Concepts and Terminology

Open-Source Component Portability Infrastructure

is a development and runtime framework embracing:

1. Open Source Software *and* Gateware
2. Component-Based Development (CBD)
3. Heterogeneous Embedded Computing

Component-Based Embedded Systems & Applications



What is OpenCPI?

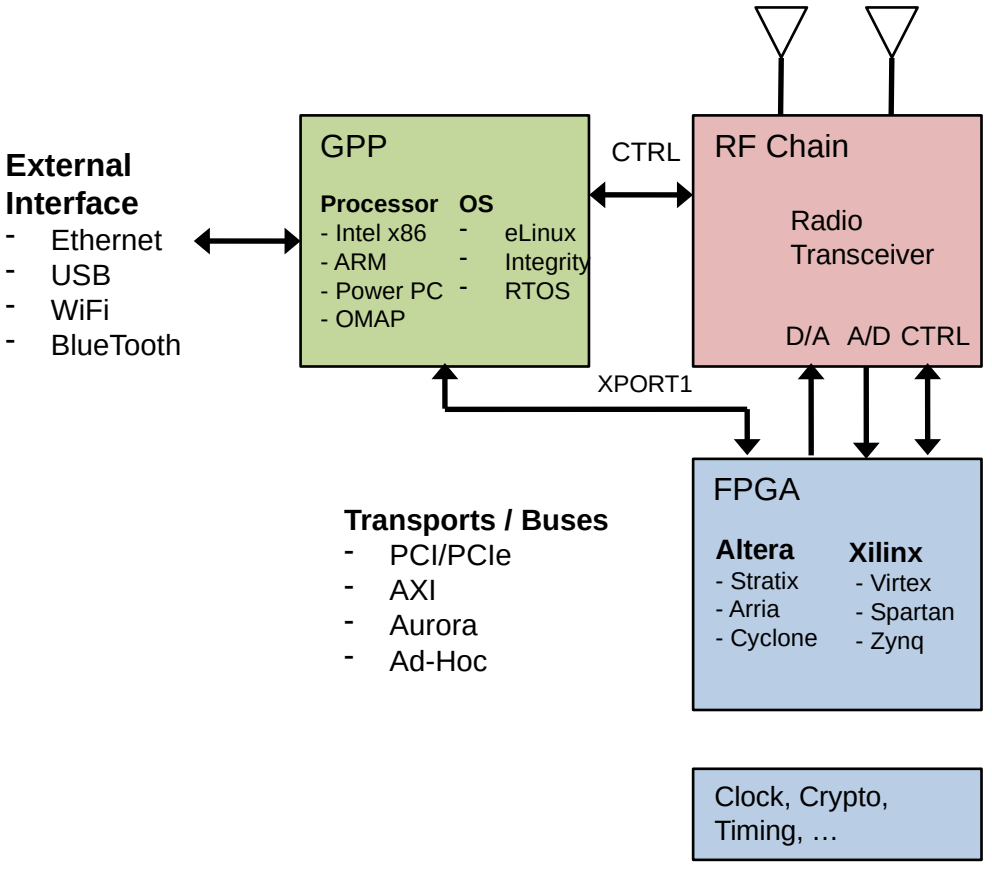
- A framework and methodology for:
  - Component-based Development
  - Mixing processor types (GPP, FPGA, GPU, DSP, Many-Core, etc.)
  - Diverse interconnect technologies (PCIe, AXI, Ethernet, Aurora)
  - Targeting embedded systems and applications
- Treating FPGA and GPU (etc.) development as a *peer* of Software
  - At the component level, not wrapped in software
- Close to hardware, with drivers (in the FPGA) that touch hardware
  - Analogous to Linux, with its hardware drivers and vendor/hardware independence
  - Providing the OS-like infrastructure on FPGAs, some drivers for Linux (SW)
  - Uses FPGA vendor tools and simulators
- OpenCPI sits below modeling tools and IDEs, it requires neither
- Cooperating/layered with a variety of other middlewares
  - ZMQ, CORBA, DDS

What does OpenCPI consist of?

- The kit is in a public open source repository on gitlab.com
- For runtime: *i.e. executing components and moving data among them*
  - Runtime libraries (software and FPGA IP) and command line utilities
  - "Drivers" (FPGA and SW) for diverse platforms, devices, interconnects
  - Portable, vendor-independent infrastructure (DMA, IO, Control, Timing)
- For development time: *i.e. developing components and applications*
  - Tools and Scripts (command line)
    - A common development workflow for FPGA and SW and GPU components
  - Specifications and Documentation
  - Examples and Tests
  - Optional GUI "proto-IDE"

OpenCPI Targets Embedded Systems

- Heterogeneous processing, mixing SW, FPGA, GPU etc.
- Commonly used for software radios, but not always



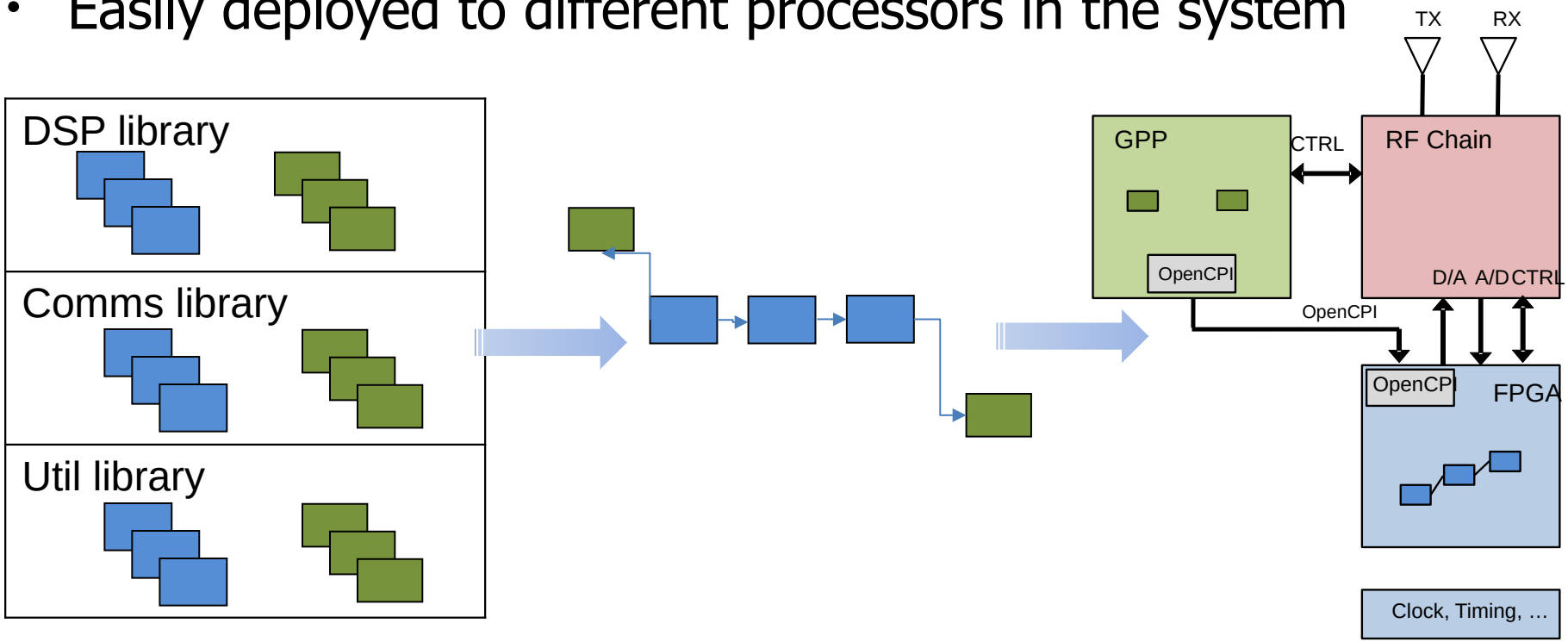
Reconfigurable Tactical Radios

Commercial SDRs

Development Platforms

OpenCPI Concept of Development

- Components developed in different languages (C++, VHDL)
- Easily deployed to different processors in the system



Portable, reusable components are developed in VHDL/C++ independent of intended application or target system

Applications are constructed from existing libraries and built (Using GUI or XML)

Application deploys to target platform with components executing on disparate parts of the system

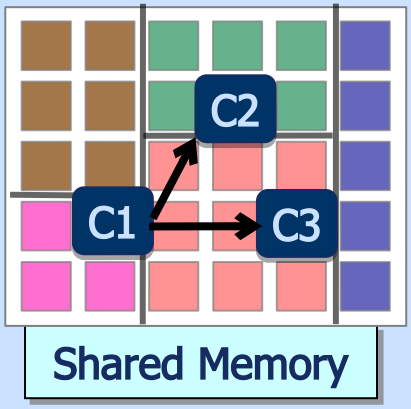
Why is OpenCPI needed?

- Especially in embedded systems, *diversity is everywhere*.
  - Meeting SWAP constraints *requires* diversity in the system solution.
  - Changes in technology or mission *require diversity over time.*
- To Support Diversity/Heterogeneity of:
  - Processing: GPP/Multicore, GPU, FPGA, DSP, many-core
    - Separate chips or SoCs (OMAP, Zynq, Arria), different vendors/architectures
  - Interconnects: 1/10GE, PCIe, SRIO, point-to-point, Infiniband, AXI…
    - And on-chip between collocated components
    - And external connections to required middleware: DDS, CORBA, ZMQ
  - Scale: Multicore SoC up to distributed or in-the-box clusters.
    - Allow embedded/tactical configurations that also work scaled up.
  - Devices: RF Transceivers, GPS, Memory, etc.
    - Many systems share the same underlying components, using same drivers
- Vendors promote their uniqueness, aspire to lock-in:
  - *They make things different that don't have to be*

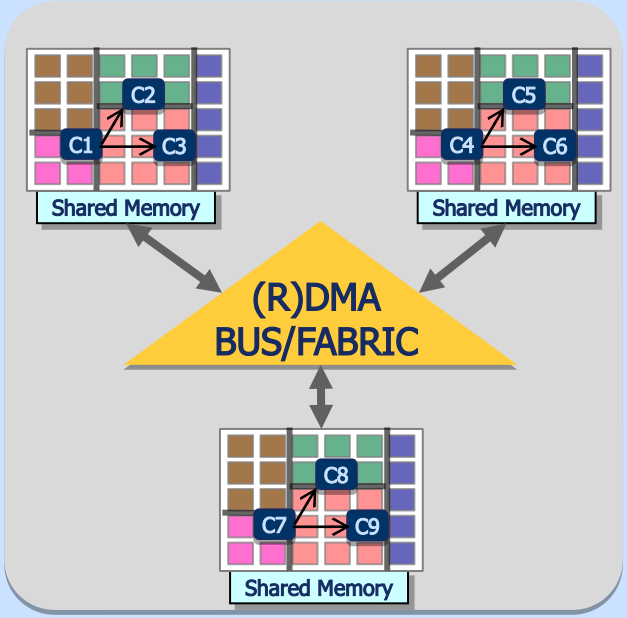
Interconnect Diversity

- Many technologies available and needed
  - *usually with different programming abstractions and APIs*

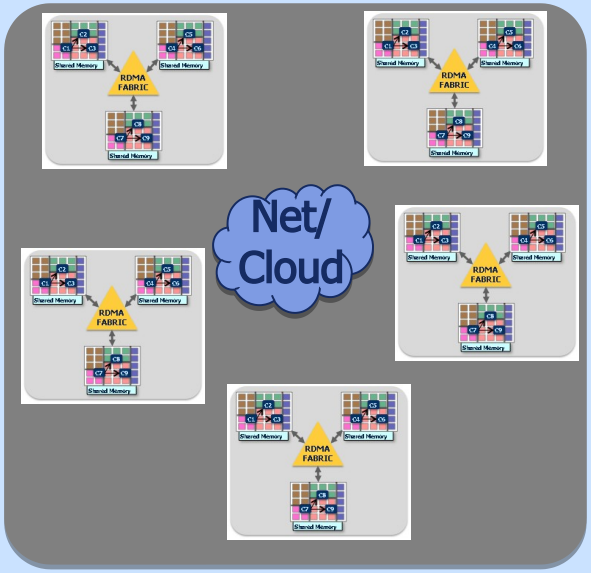
Intra-Chip/On-Chip
Multi-core/FPGA/GPU/DSP



Inter-Chip/Board/In-Box
Fabric/Bus/Backplane



Inter-Box/Clusters
Network (IP/TCP/UDP)



Multicore Processing

Multi-node Processing

Grid/Cluster/Distributed Processing

OpenCPI Focus is Mostly Here, but Distributed Networking is Supported

OpenCPI alleviates and attacks diversity

- Commonality at a reasonable price
  - Careful, vendor and technology-neutral choices
  - Don't reinvent how algorithmic code is written (i.e. no new languages)
  - Suppress diversity without significant performance or resource tax
  - Common workflow, methodology, interfaces, reduces learning

- Open source is key:
  - Industry, products, tools are in silos, even when open-source
  - Community and openness is necessary to cross all the boundaries

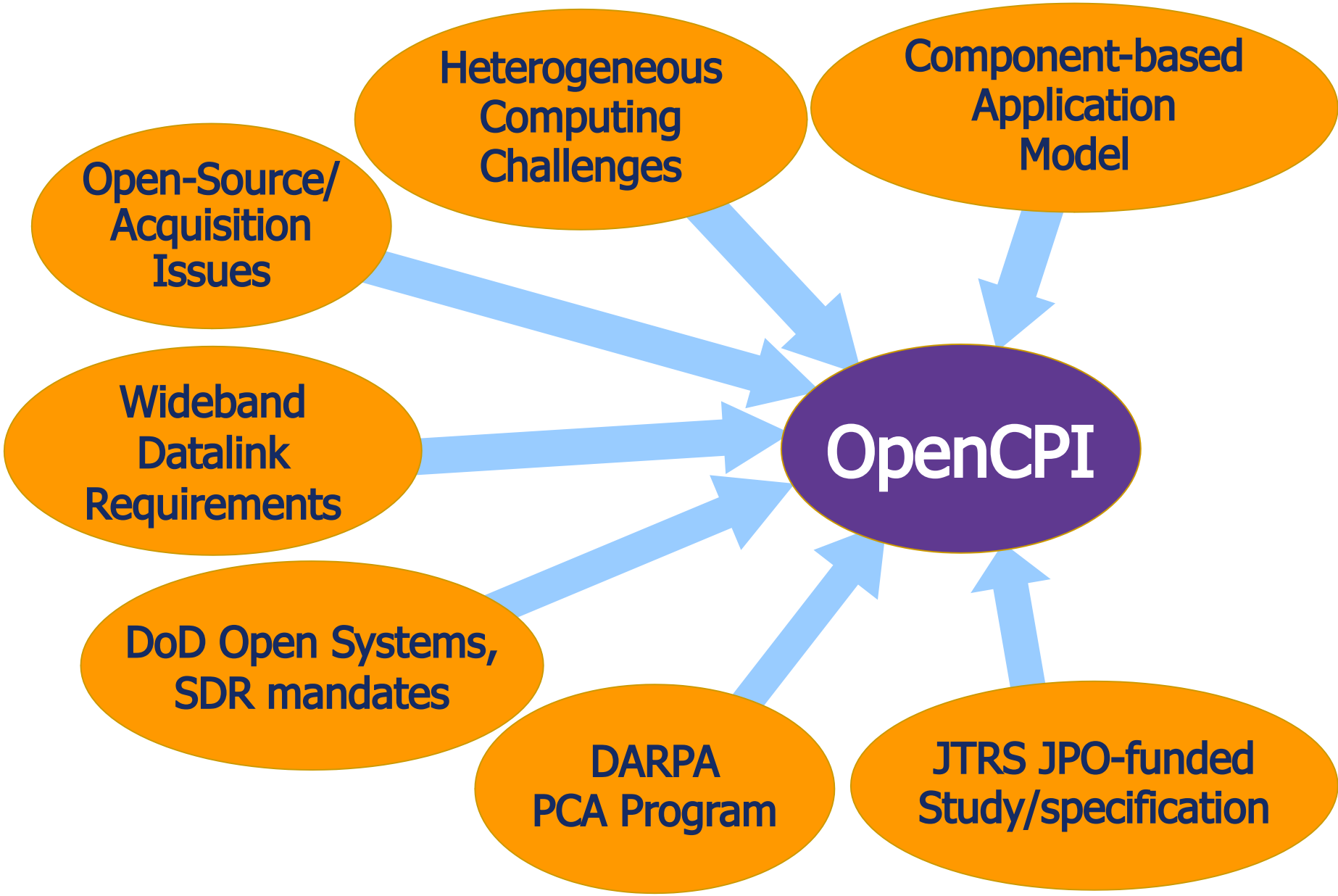
- OpenCPI addresses this diversity as its primary focus:
  - Paying attention to embedded efficiency requirements
  - Crossing boundaries not addressed by other approaches
  - Common component-based design & runtime across the boundaries
  - Reduce unnecessary differences for design, development, training

Where did OpenCPI come from?

- It evolved via a succession of IR&D and DoD funded projects
- Core motivation was: *apply CBD to real embedded applications*
- Various application areas were assessed and needs addressed
- Many lessons learned, many boundaries crossed

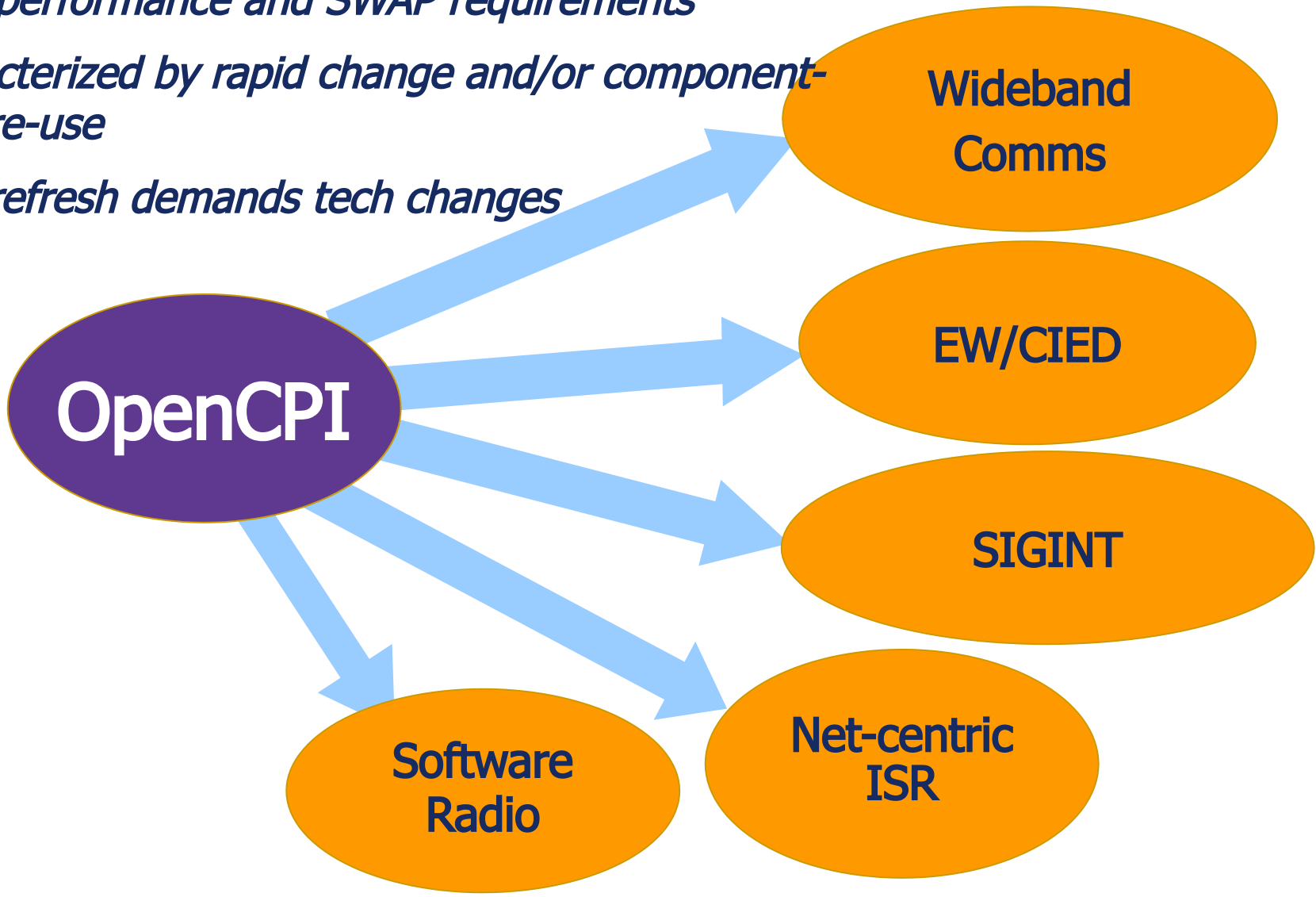
- 2000-2002 Mercury Computer IR&D for Broadcast Video/Medical
- 2002-2007 DARPA Polymorphous Computing Architecture program
- 2004-2005 JTRS Program Office: SDR/SCA for DSP & FPGA
- 2007-2009 AF HDR-RF SATCOM Modem (w/Raytheon)
- 2008-2010 JEIDDO/I2WD JCREW S&T (Counter-IED)
- 2009-2010 AFRL/DDR&E ***Transition To Open Course***
- 2010-2015 AFRL/DDR&E Software Producibility Initiative
- 2012- DoD Embedded Platform Framework (various)

Where OpenCPI comes from



Where OpenCPI can be applied:

- *Application areas using embedded technologies to meet performance and SWAP requirements*
- *Characterized by rapid change and/or component-level re-use*
- *Tech refresh demands tech changes*



Positioning and Priorities

- OpenCPI is a development framework that cuts across technologies and vendors, providing "insulation" from diversity
  - GPP, FPGA, GPU etc.
  - Xilinx, Intel/Altera, Mentor, etc.
  - Nvidia, AMD/ATI, Intel, etc.
  - ADI, Lime, other ADC/DAC/ GPS devices
- High priorities:
  - reduce vendor-specific training, tools, methods
  - extreme openness – access to all source code
    - rarely get stuck unable to fix a problem or hire your own expert
    - *move the "openness" frontier closer to hardware*
    - don't preclude proprietary modules when they are indeed modular
  - facilitate change: *reduce time and effort to move embedded apps*
    - changing processing technologies, and devices
    - changing vendors
    - changing algorithms
    - changing configurations
    - changing systems and platforms

- Definition, Positioning, Priorities, Background
- Component Concepts and Terminology
- Application Concepts and Terminology

Component Model: What's a component?

Name/function: What does it do? e.g. "encrypt"

Properties: How is it controlled/configured?

- What are the switches, knobs, meters, parameters?

Ports: How does it communicate its input and output data?

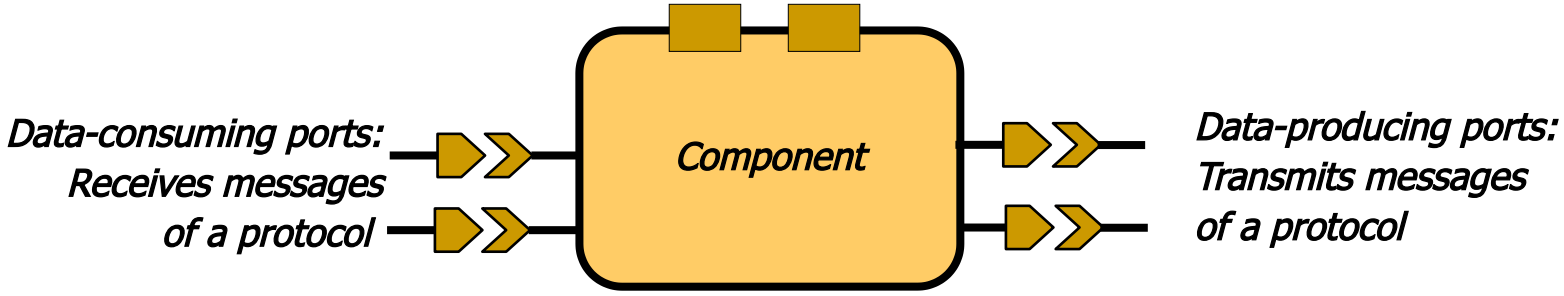
- With *protocols* that say what messages come in or go out

Execution Model:

- Autonomous, simplified "actor" model, inherently concurrent, interacting with others via messages at connected ports

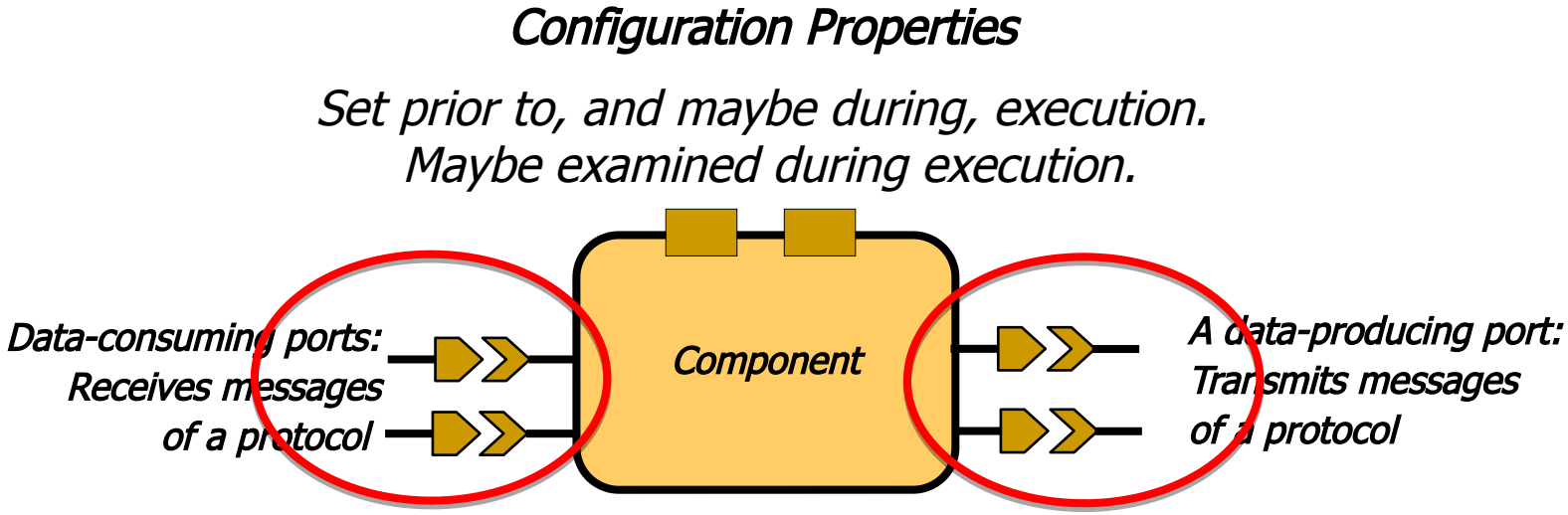
Configuration Properties

Set prior to, and maybe during, execution.
Maybe examined during execution.



Communication Model: What's a protocol?

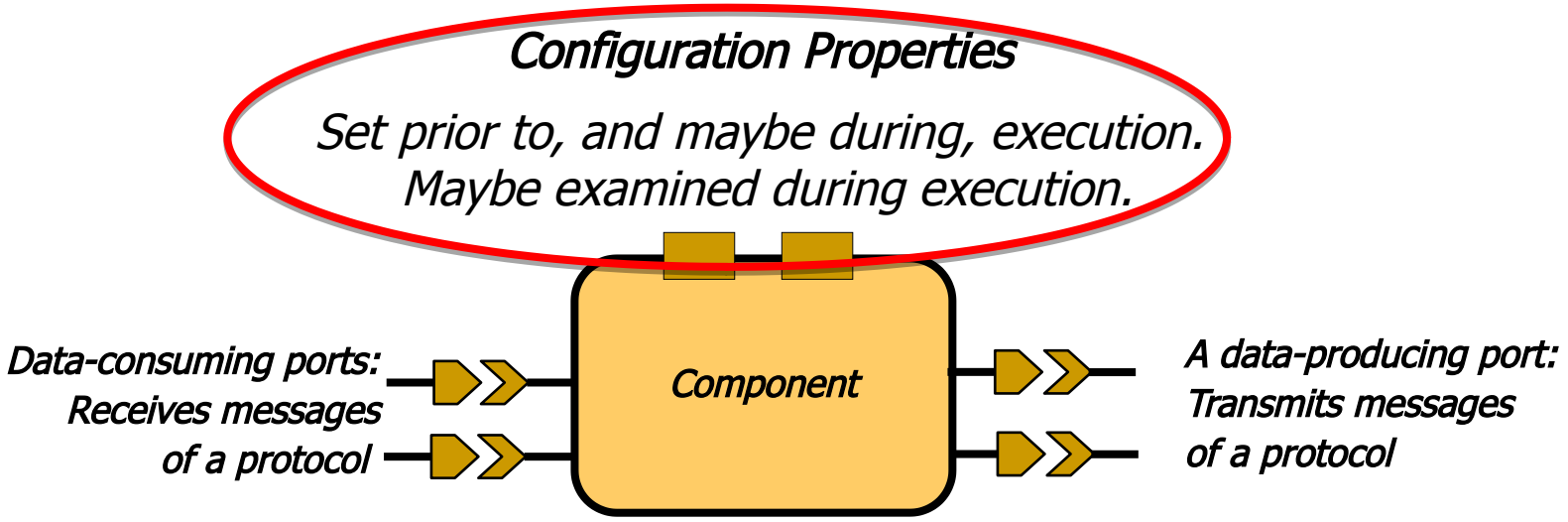
- A component port usually has an associated protocol.
- An OpenCPI protocol is simply a set of defined messages.
  - Many "protocols" are a single type of message
  - Many messages are simply a fixed or variable length array of scalars
  - Messages can be zero length (an event with no associated data)
  - Messages are defined with data fields of various types
- Each port specifies what protocol (what messages) it supports.
  - Connections between component ports are valid if the protocols match



Control Model: Lifecycle and Properties

- Components are managed through a standard "lifecycle"
  - Initialize, start, stop, shutdown, etc.

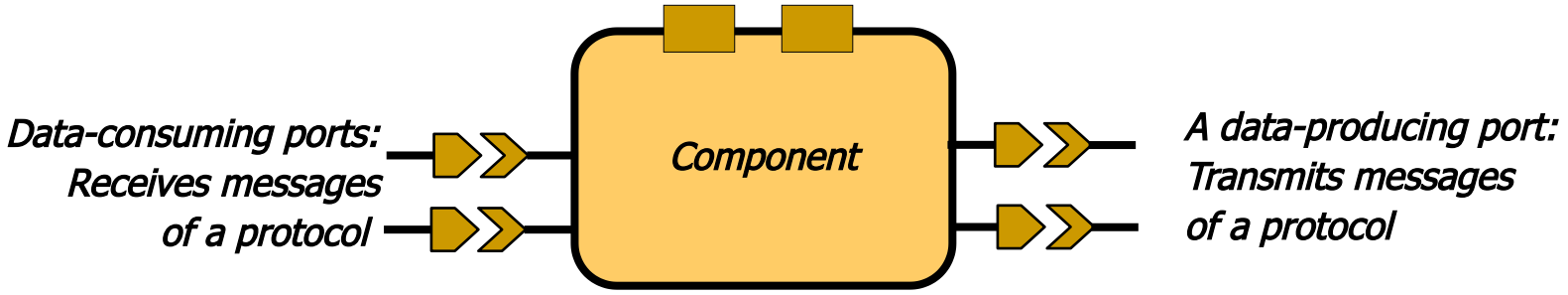
- Properties are the "knobs and meters" of a component
  - Some are for "initial configuration"
  - Some are for "dynamic control"
  - Some are for feedback (e.g. a peak meter)



Component Specifications vs. Implementations

- Components are defined functionality
  - E.g. data compression, signal processing, etc.
  - With properties and ports (with protocols)
  - Specified in a small XML document – a *component spec*
    - that may refer to *protocol specs* common to different components' ports

- A component can have many implementations: *workers*
  - Written in different languages to OpenCPI-defined APIs
  - Compiled/synthesized for different targets (CPUs, FPGAs etc.)
  - All meet the same component spec, perform the same function
  - Perhaps using different algorithms to do the same job differently

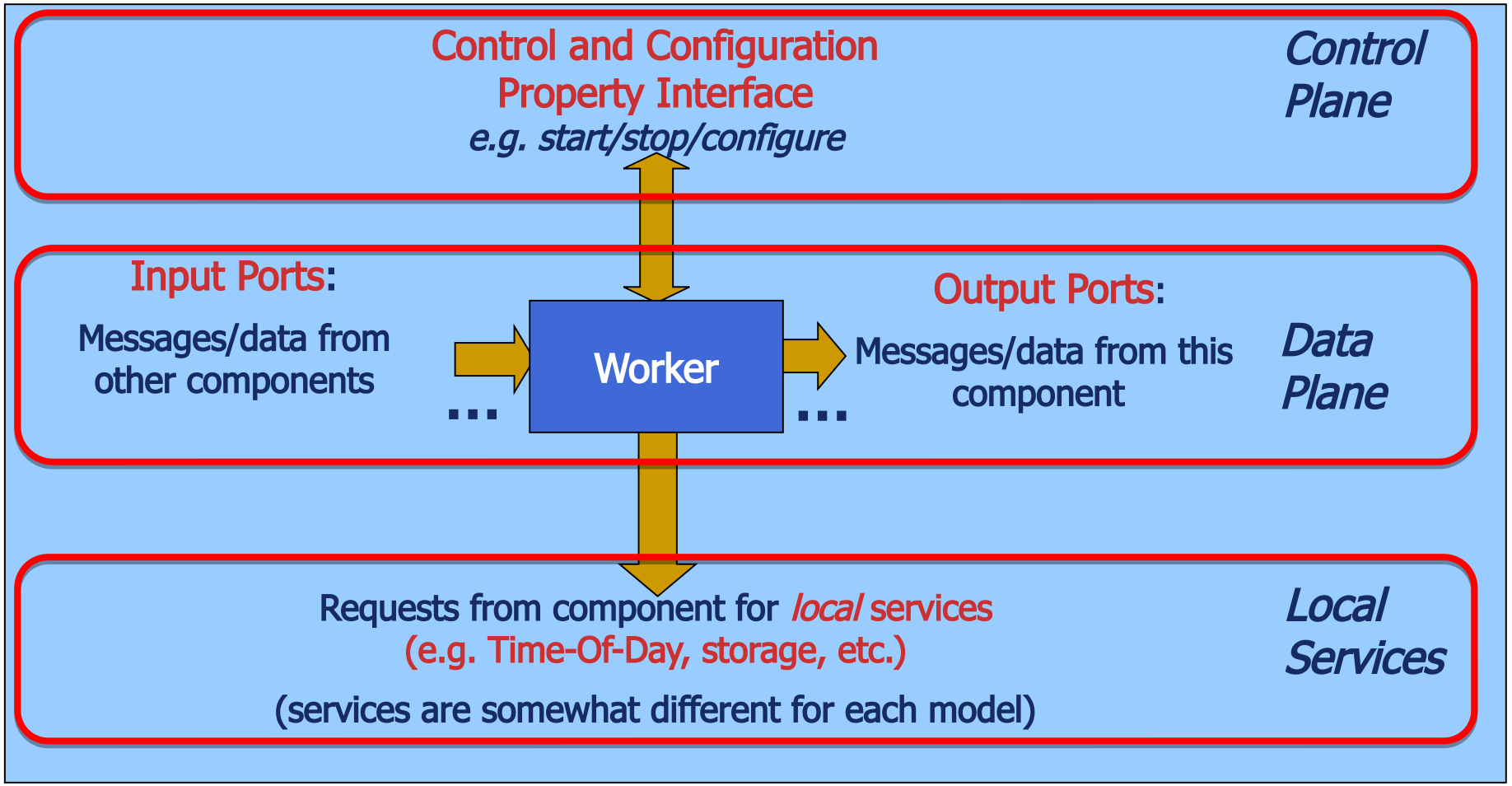


Authoring Models: How to Write Workers

- A component can be implemented by different *workers*
- Worker source code is written for an *authoring model*
- An Authoring Model:
  - Defines one of several alternative ways to write a worker
    - *Usually for languages well suited for some processing technology*
  - It defines the interfaces (APIs) for control and communication (data)
  - It must fit well with:
    - *How practitioners actually write code for some technology*
    - *Tools/compilers that enable effective use of the technology*
    - *Simple enough for technology-appropriate implementations*
  - Must coexist and interoperate with workers from other models
  - Must efficiently interact with collocated similar workers
- OpenCPI has authoring models for GPP, FPGA, GPU, DSP

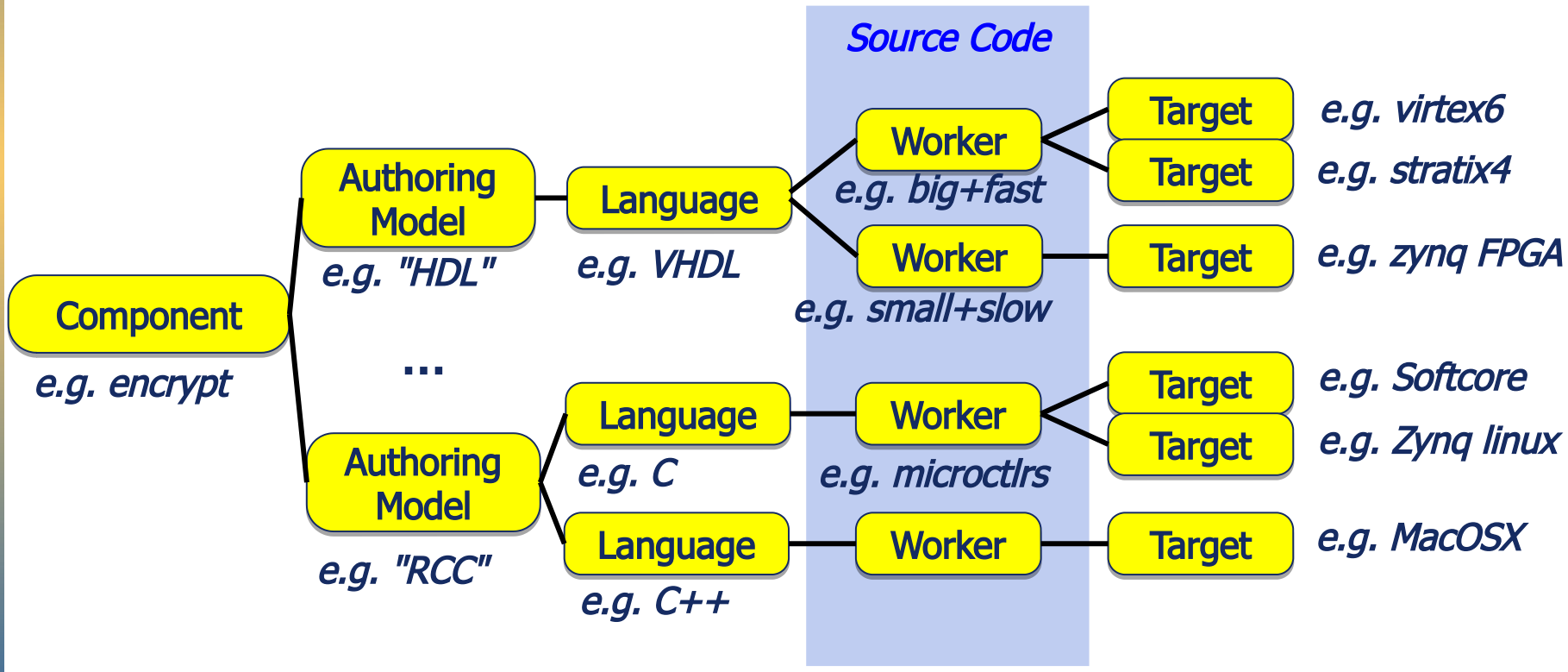
Authoring Models: The Pattern for Each One

Define interfaces (APIs) and behavior for three aspects:



Implementing Components

- Workers (component implementations) are written:
  - Implementing a *component specification*
  - Using an *authoring model*
  - In a *language* defined for that authoring model
  - Compiled for some *target* processors and environments



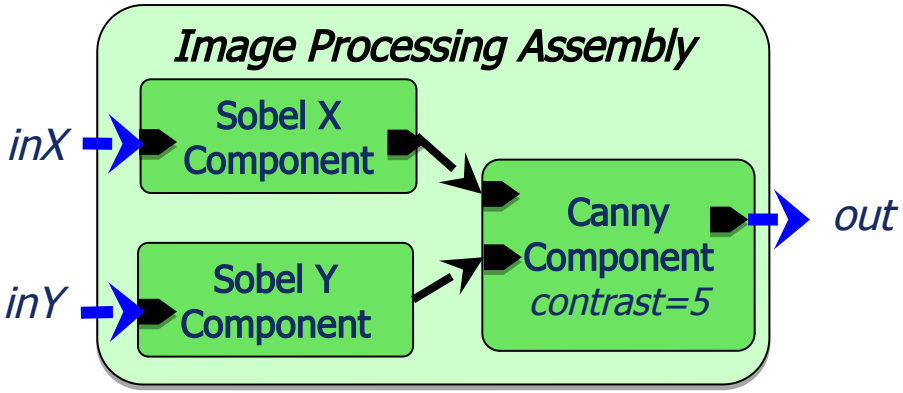
OpenCPI Terms for Components

- Component: building blocks for applications
  - Defined in XML as a "spec" for (alternative) implementations
- Property: component configuration and control
  - Defined as part of component "spec", usable at runtime
- Port: component interfaces to talk to others
  - Defined as part of component "spec" to send/receive messages
- Protocol: the set of supported messages at a port
  - Defined in XML and associated with component ports
- Authoring Model: one of several ways to write worker code
  - For different technologies, in different languages, different APIs
- Worker: one of several *implementations* of a component
  - Defined in XML and associated with a component, *plus source code*

- Definition, Positioning, Priorities, Background
- Component Concepts and Terminology
- Application Concepts and Terminology

What's an OpenCPI Application?

- An interconnected set of ***components,*** called an ***assembly***



- Each ***component*** in the ***assembly*** indicates that, at runtime,
  - *Some* implementation (worker binary) must be found for this component
  - A executing runtime-instance must be created from that binary
- Each component can be assigned ***initial property values***
- Some ports are identified as ***external ports*** of the assembly
  - Suggestive of, but not a true hierarchy
  - Can be bound to files or other I/O devices when the application is run

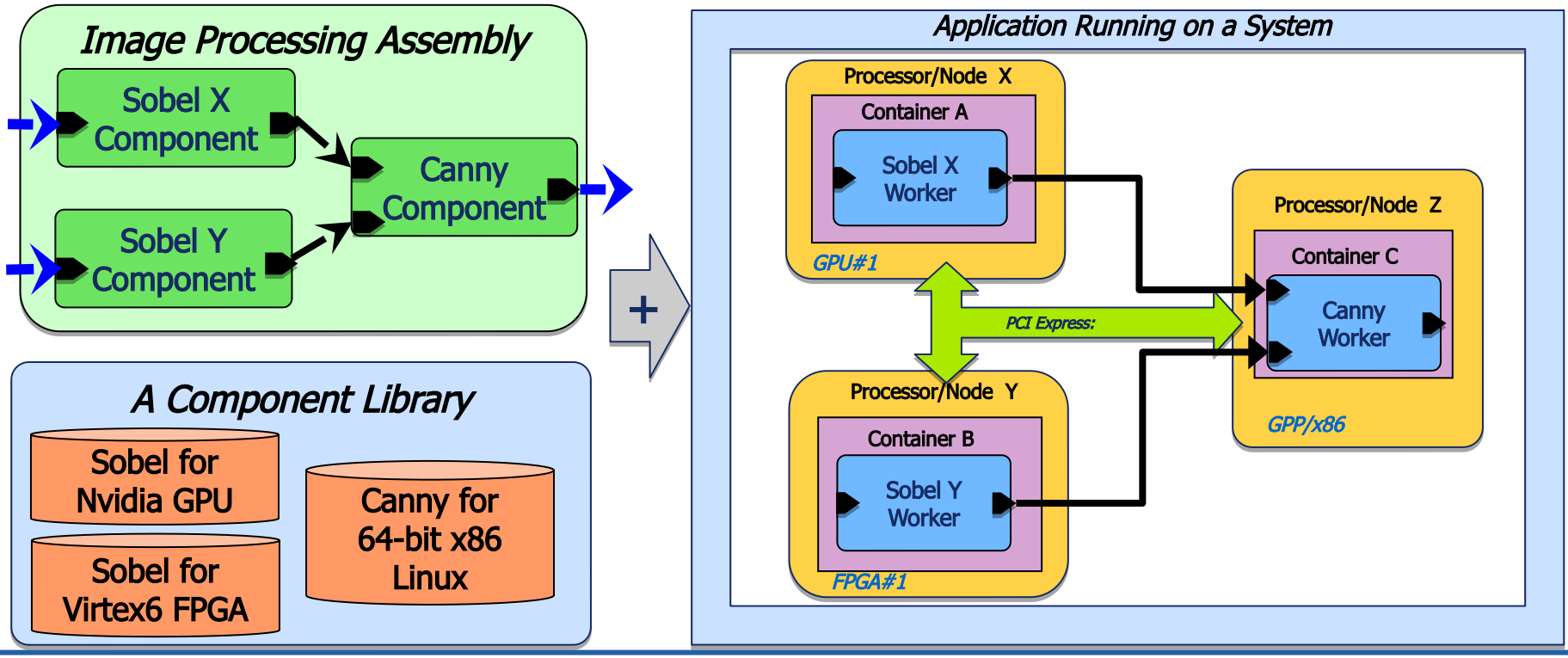
OpenCPI Terms for *Defining* Applications

How We Define Applications

- Assembly: a set of components with interconnected ports
  - Defined in XML (possibly program-generated) and used to launch applications
- Property Values: the initial set of values set at startup
  - Associated with the components in the assembly
- External Ports: the component ports for external I/O
  - Indicated in the assembly for specific component ports
  - Allows these to be bound externally to other sources/sinks of data

How do we run an application?

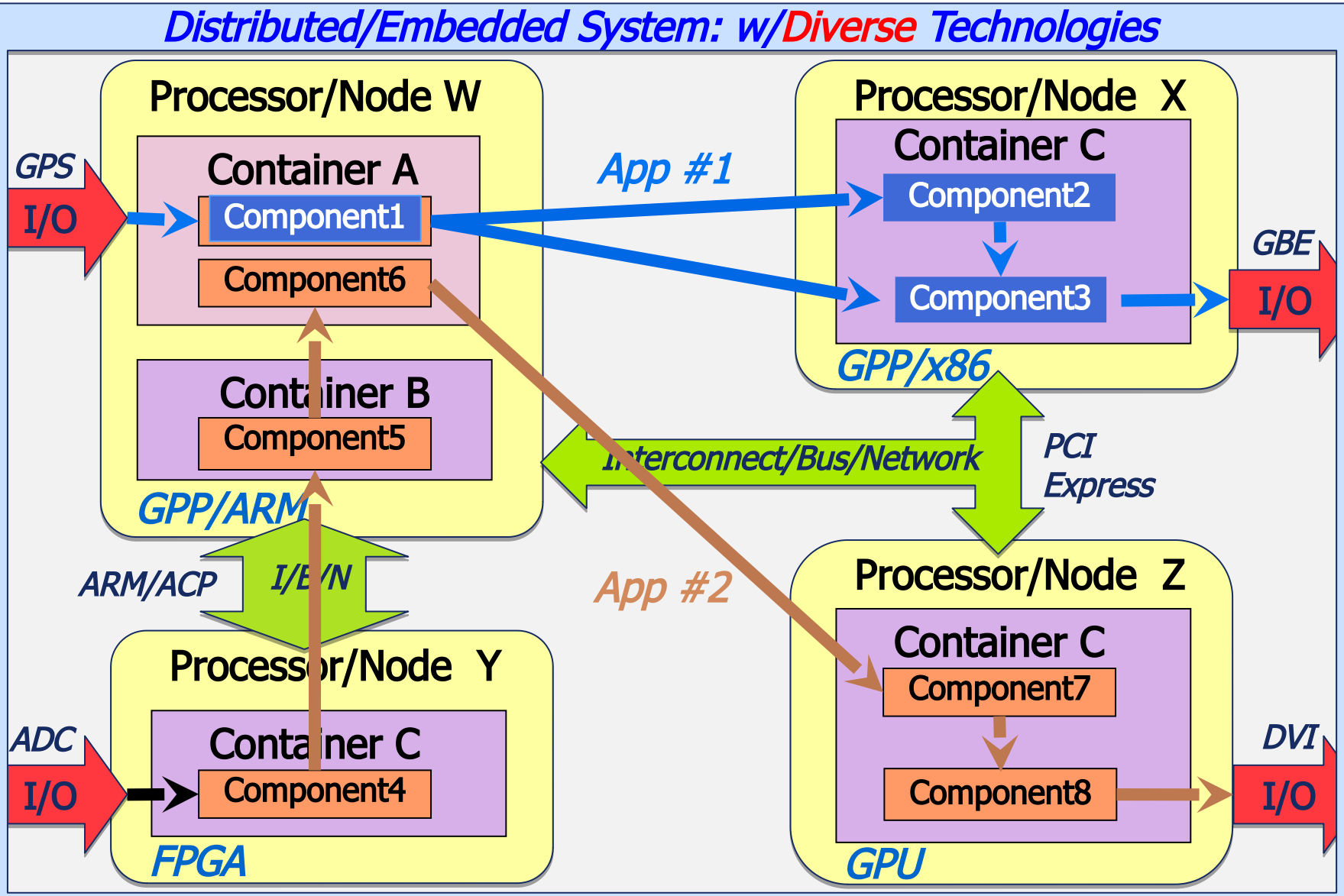
- For each component in the assembly, the system:
  - Finds a binary *artifact* containing an implementation (compiled *worker*)
  - Finds a runtime environment (*container*) where the *worker* can run
- *Artifacts* (compiled binaries) are found in (heterogeneous) *libraries*.
  - Artifacts are loaded on demand into the containers to run the assembly
- Execution requires: assembly + artifact libraries + containers



OpenCPI Terms for Executing Applications

- Artifact: compiled binary package of one or more workers
  - Result of building/compiling some workers (source code)
  - Includes embedded metadata needed for runtime discovery
  - *Physical unit that can be patched, updated, emailed, like a DLL/so.*
- Artifact Library: collections of artifacts for runtime
  - Typical "library" concept
  - Runtime has a typical "library path" to find artifacts from libraries
- Container: the execution environment for workers
  - Managing the execution of workers on a processor
  - Arranging control and data communications with other containers
  - Typically on a "target processing node"

Component-Based Embedded Systems & Applications



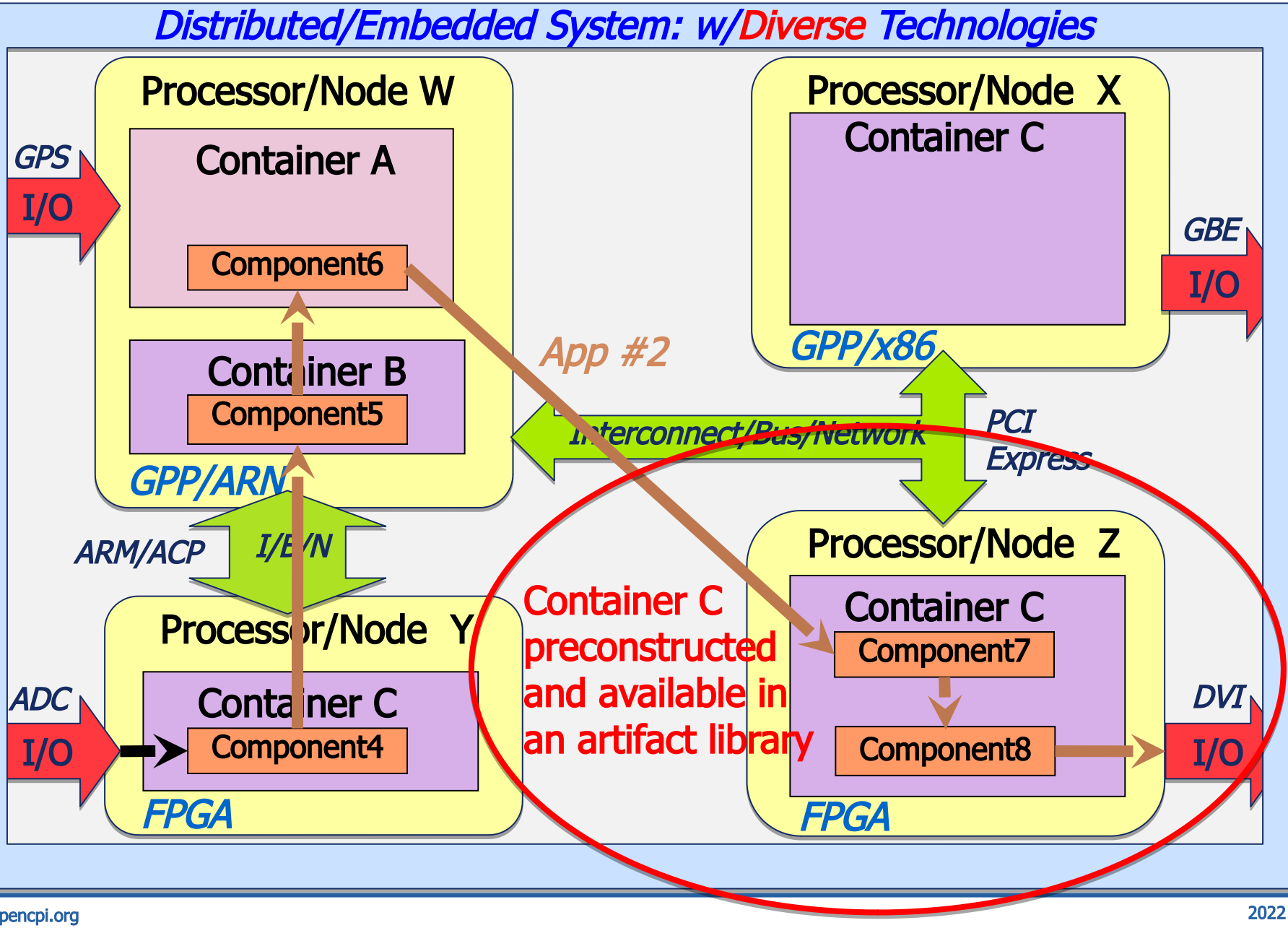
FPGA Developers: How Can FPGAs Work This Way?

- FPGA "processors" are not so dynamic
- OpenCPI is *not* inventing on-the-fly constructed FPGA bit files
  - Although this does work for simulators
- OpenCPI is *not* relying on "partial reconfiguration"
  - Although it is a natural fit for this model, and on the roadmap

- OpenCPI ***artifacts*** for FPGAs are:
  - Pre-constructed ***subassemblies***
    - *A set of connected workers with external ports*
    - *Some DSPs have similar statically linked configurations*

- So, if an FPGA ***artifact*** can implement a *subset* of the application, then that artifact can be used on an FPGA
  - Artifacts contain embedded metadata describing what is there

- OpenCPI *does* exploit dynamic/rapid (re)loading of FPGAs


Distributed/Embedded System: w/Diverse Technologies
Processor/Node W
Container A
Component6
Container B
Component5
GPP/ARM
GPS
I/O
ARM/ACP
I/E/N
Processor/Node Y
Container C
Component4
FPGA
ADC
I/O
App #2
Interconnect/Bus/Network
PCI Express
Processor/Node X
Container C
GPP/x86
GBE
I/O
Container C preconstructed and available in an artifact library
Processor/Node Z
Container C
Component7
Component8
FPGA
DVI
I/O

Open-Source Component Portability Infrastructure

is a development and runtime framework embracing:

1. Open Source Software *and* Gateware
2. Component-Based Development (CBD)
3. Heterogeneous Embedded Computing