"Multicore programming"

No more communication in your program, the key to multi-core and distributed programming.

Eric.Verhulst@OpenLicenseSociety.org

Commercialised by:

Content

- About Moore's imperfect law
- The von Neuman syndrome
- Why multicore, it is new?
- Where's the programming model?
- The OpenComRTOS approach:
  - Formally modeled
  - Hubs and packet switching
  - Small code size
  - Virtual Single Processor model
  - Scalability, portability, ...
  - Visual Programming

Moore's law

- Moore's law:
  - Shrinking semicon features => more functionality and more performance
- Rationale: clock speed can go up
- The catch is at system level:
  - Datarates must follow
  - Memory access speed must follow
  - I/O speeds must follow
  - Througput (peak performance vs. latency (real-time behaviour)
  - Power consumption goes up as well (F^2, Vcc)
- => Moore's law is not perfect

The von Neuman syndrome

- Von Neuman's CPU:
  - First general purpose reconfigurable logic
    - Saves a lot of silicon (space vs. time)
    - Separate silicon architecture from configuration
      - "program" in memory => "reprogrammable
      - CPU state machine steps sequentially through program
  - The catch:
    - Programming language reflects the sequential nature of the von Neuman CPU
    - Underlying hardware is visible (C is abstract asm)
    - Memory is much slower than CPU clock
      - PC: > 100 times! (time to do 99 other things while waiting
    - Ignores real-world I/O
    - Ignores that software are models of some (real) world
    - Real world is concurrent with communication and synchronisation

Why Multi-Core?

- System-level:
  - Trade space back for time and power:
    - 2 x F > 2*F, when memory is considered
  - Lower frequency => less power (~1/4)
  - Embedded applications are heterogous:
    - Use function optimised cores
- The catch:
  - Von Neuman programming model incomplete
  - Distributed memory is faster but
    - requires “Network-On- and Off-Chip”

Multi-Core is not new

- Most embedded devices have multi-core chips:
  - GSM, set-up boxes: from RISC+DSP to RISCs+DSPs+ASSP+... = MIMD
  - Not to be confused with SMP and SIMD
- Multi-core = parallel processing (board or cabinet level) on a single chip
- Distributed processing widely used in control and cluster farms
- The new kid in town = communication
  - (on the chip)

Where's the (new) programming model?

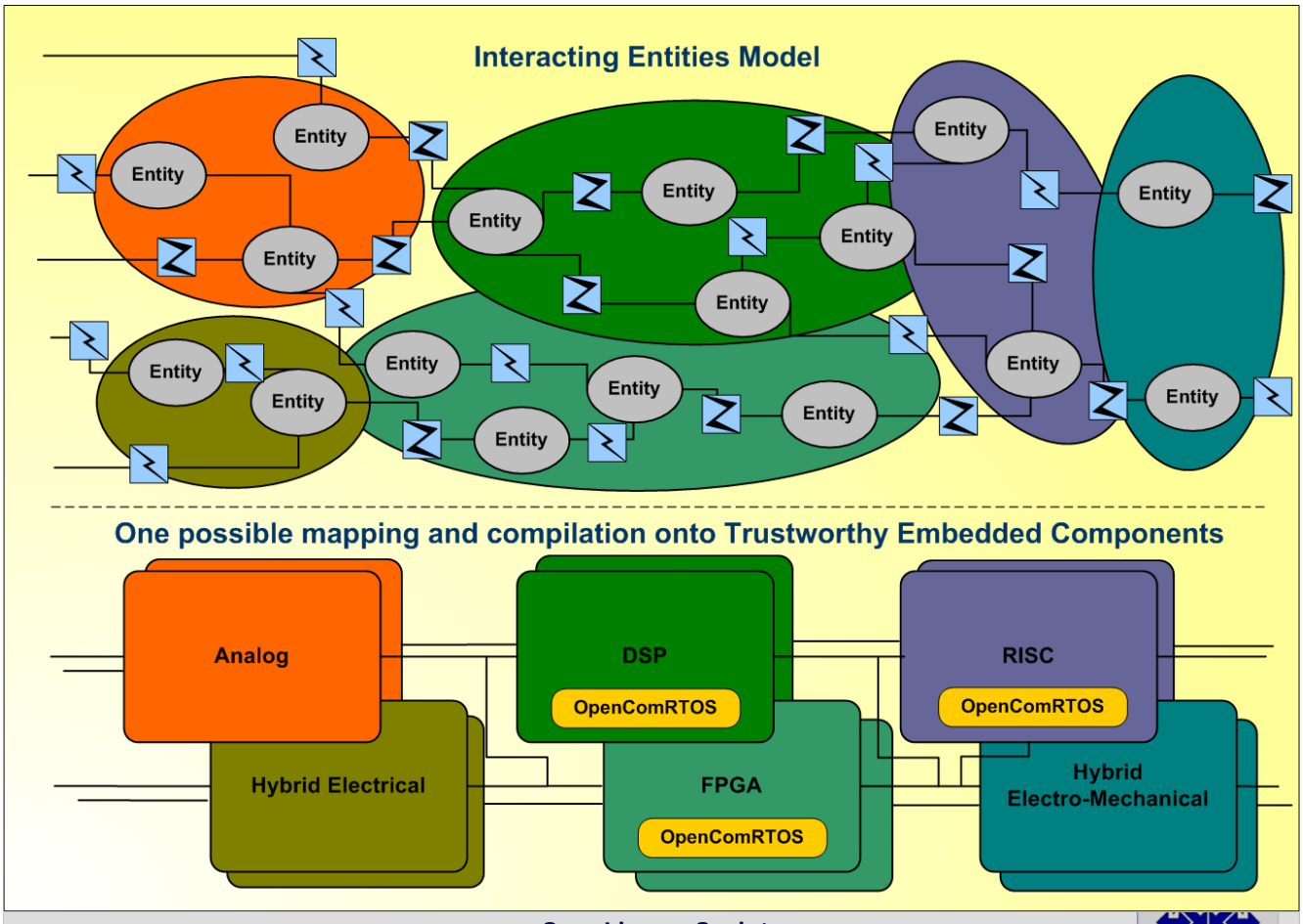
- Issue: what about the "old" software?
  - => von neuman => shared memory syndrome
  - But: issue is not access to memory but integrity of memory
  - But: issue is not bandwidth to memory, but latency
  - Sequential programs have lost the information of the inherent (often async) parallelism in the problem domain
- Most attempts (MPI, ...) just add a large communication library:
  - Issue: underlying hardware still visible
  - Difficult for:
    - Porting to another target
    - Scalability (from small to large AND vice-versa)
    - Often application domain specific
    - Performance doesn't scale

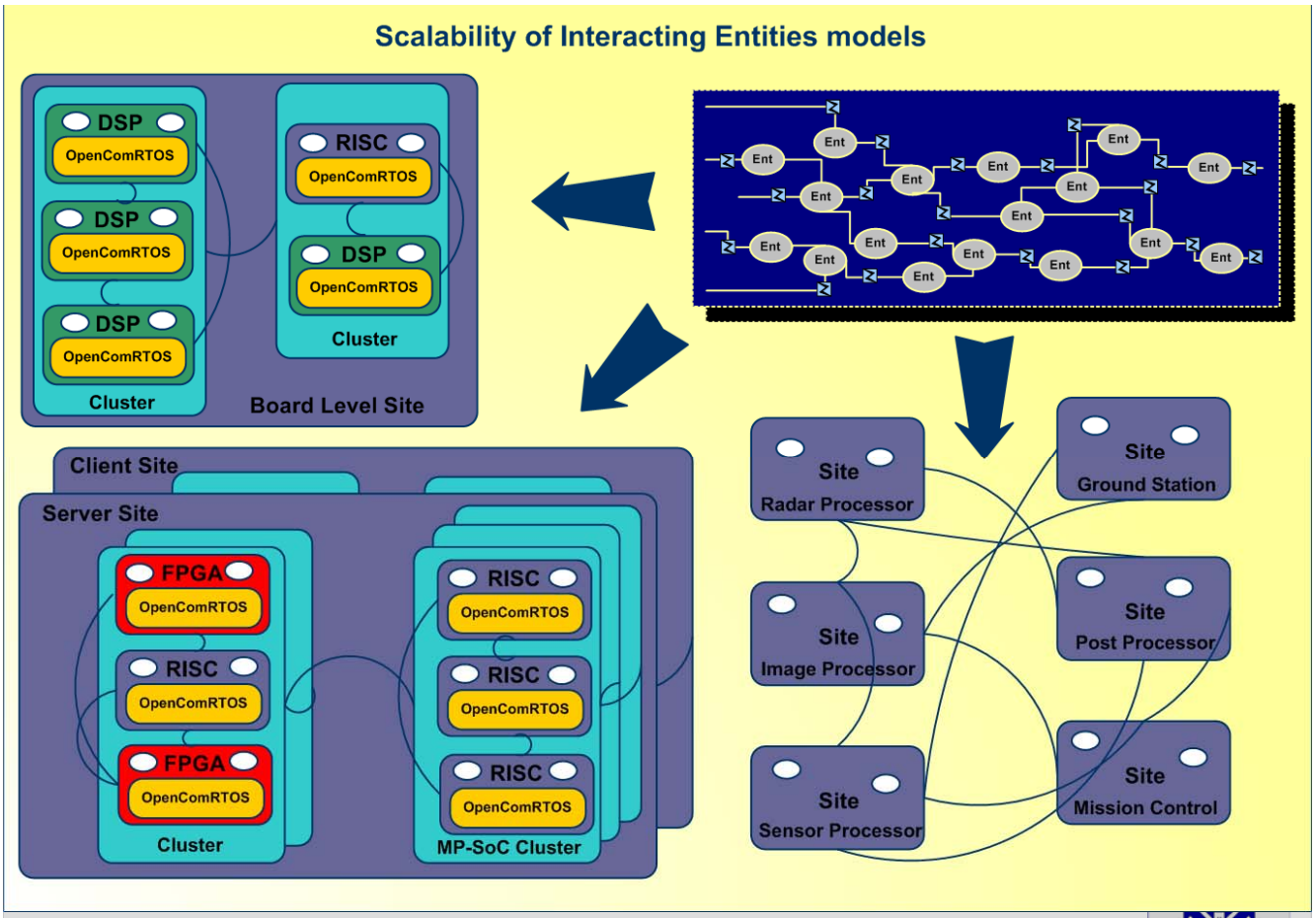
The OpenComRTOS approach

- Derived from a unified systems engineering methodology
- Two keywords:
  - Unified Semantics
    - use of common "systems grammar"
    - covers requirements, specifications, architecture, runtime, ...
  - Interacting Entities (models almost any system)
- RTOS and embedded systems:
  - Map very well on "interacting entities"
  - Time and architecture mostly orthogonal
  - Logical model is not communication but "interaction"

The OpenComRTOS project

- Target systems:
  - Multicore, parallel processors, networked systems, include "legacy" processing **nodes** running old (RT)OS
- Methodology:
  - Formal modeling and formal verification
- Architecture:
  - Target is multi-node, hence communication is system-level issue, not a programmer's concern
  - Scheduling is orthogonal issue
  - An application function = a "task" or a set of "tasks"
    - Composed of sequential "segments"
  - In between:
    - Tasks synchronise and pass data ("interaction")

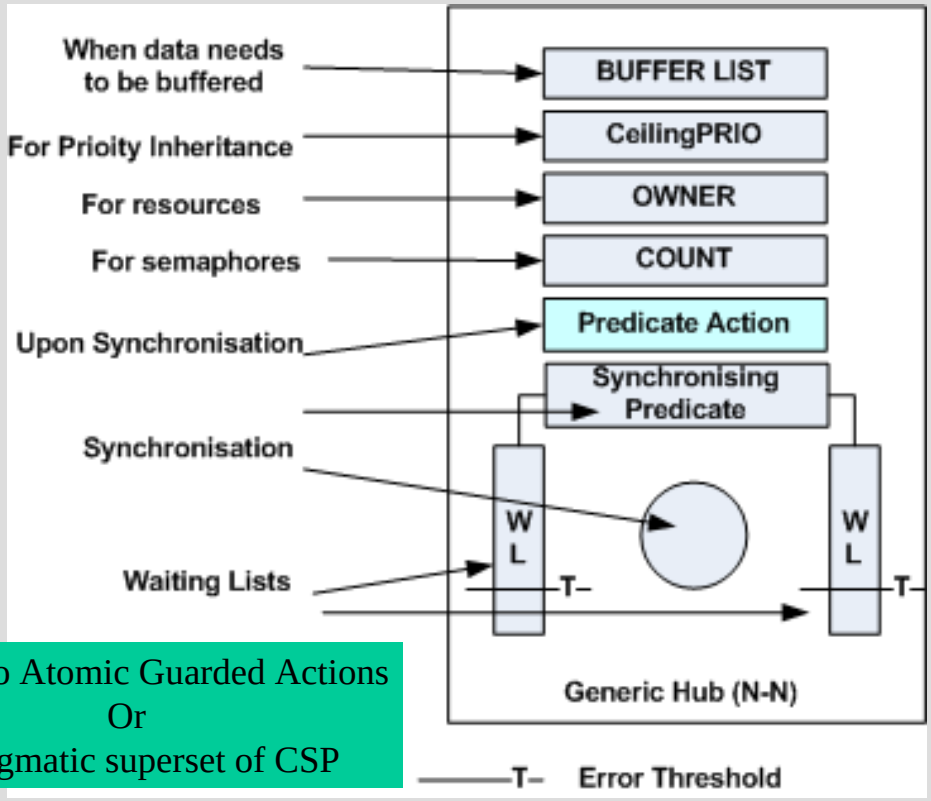




The OpencomRTOS "HUB"

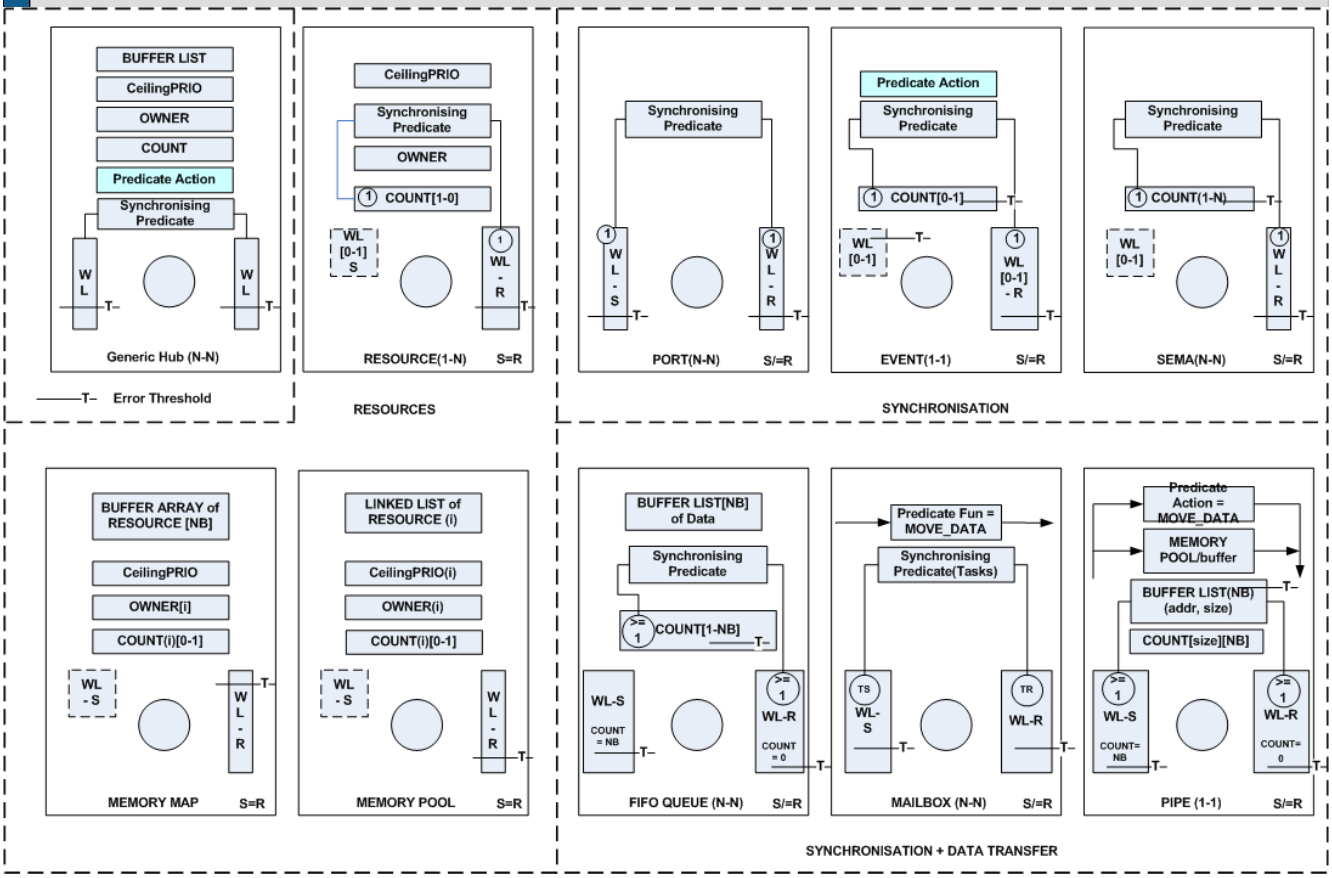
- Result of formal modeling
- Events, semaphores, FIFOs, Ports, resources, mailbox, memory pools, etc. are all variants of a generic HUB
- A HUB has 4 functional parts:
  - Synchronisation point between Tasks
  - Stores task's waiting state if needed
  - Predicate function: defines synchronisation conditions and lifts waiting state of tasks
  - Synchronisation function: functional behavior after synchronisation: can be anything, including passing data
- All HUBs operate system-wide, but transparently: Virtual Single Processor programming model
- Possibility to create application specific hubs & services! => a new concurrent programming model

Graphical view of RTOS “Hubs”

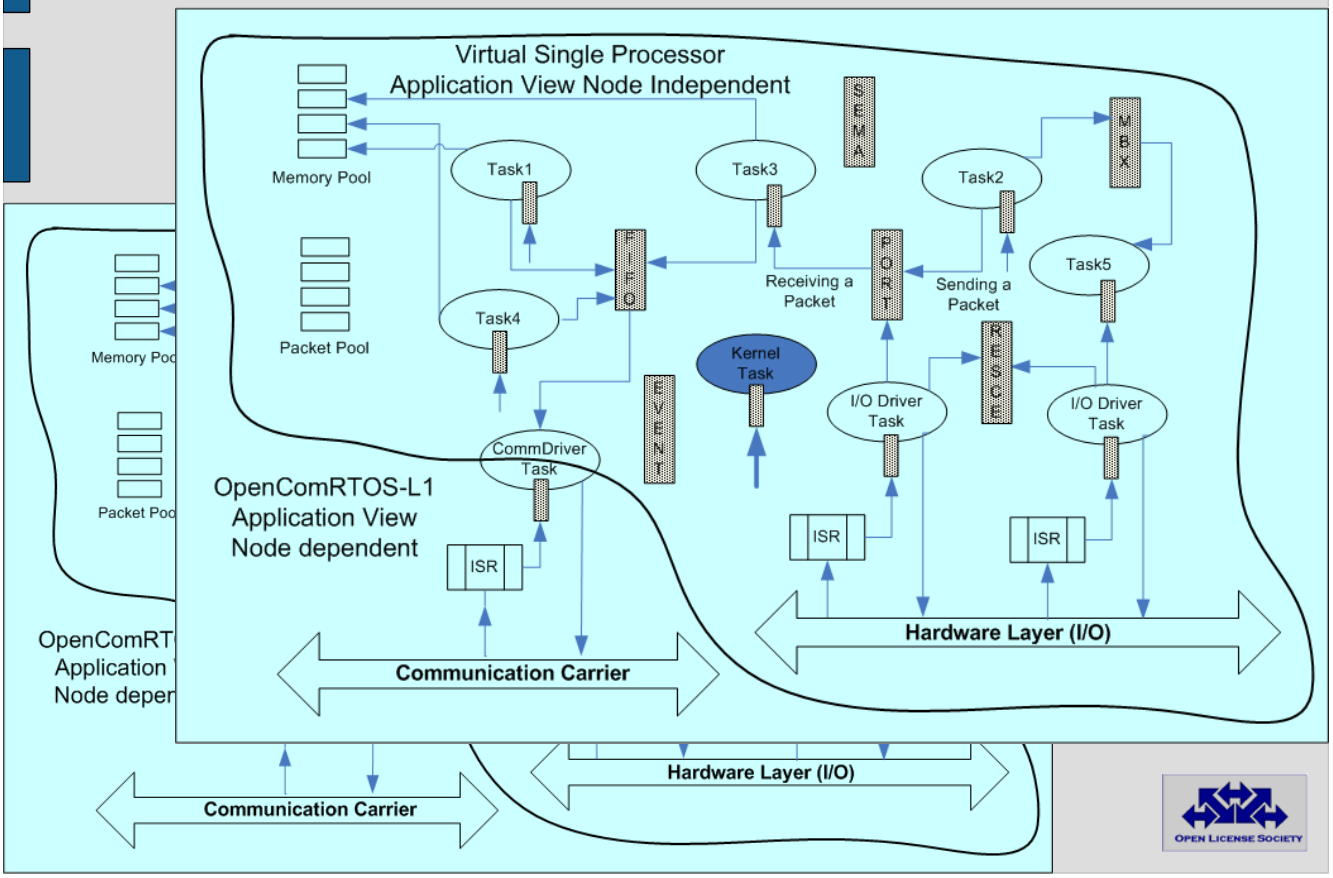


Similar to Atomic Guarded Actions
Or
A pragmatic superset of CSP

All RTOS entities are “HUBs”



L1 application view:
any entity can be mapped onto any node



Rich semantics: _NW|W|WT|Async

- L1_Start/Stop/Suspend/ResumeTask
- L1_SetPriority
- L1_SendTo/ReceiveFromHub
- L1_Raise/TestForEvent_(N)W(T)_Async
- L1_Signal/TestSemaphore_X
- L1_Send/ReceivePacket_X L1_WaitForAnyPacket_X
- L1_Enqueue/DequeueFIFO_X
- L1_Lock/UnlockResource_X
- L1_Allocate/DeallocatePacket_X
- L1_Get/ReleaseMemoryBlock_X
- L1_MoveData_X
- L1_SendMessageTo/ReceiveMessageFromMailbox_X
- L1_SetEventTimerList
- ... => user can create his own service!

Unexpected: RTOS 10x smaller

- Reference is Virtuoso RTOS (ex-Eonic Systems)
- New architectures benefits:
  - Much easier to port
  - Same functionilaty (and more) in 10x less code
  - Smallest size SP: 1 KByte program, 200 byts of RAM
  - Smallest size MP: 2 KBytes
  - Full version MP: 5 KBytes
- Why is small better ?
  - Much better performance (less instructions)
  - Frees up more fast internal memory
  - Easier to verify and modify
- Architecture allows new services without changing the RTOS kernel task!

Clean architecture gives small code: fits in on-chip RAM

OpenComRTOS L1 code size figures (MLX16)				
	MP FULL		SP SMALL	
	L0	L1	L0	L1
L0 Port	162		132	
L1 Hub shared		574		400
L1 Port		4		4
L1 Event		68		70
L1 Semaphore		54		54
L1 Resource		104		104
L1 FIFO		232		232
L1 Resource List		184		184
Total L1 services		1220		1048
Grand Total	**3150**	**4532**	**996**	**2104**

Smallest application: 1048 bytes program code and 198 bytes RAM (data)
(SP, 2 tasks with 2 Ports sending/receiving Packets in a loop, ANSI-C)
Number of instructions : 605 instructions for one loop (= 2 x context switches, 2 x L0_SendPacket_W, 2 x L0_ReceivePacket_W)

Probably the smallest MP-demo in the world

	Code Size	Data Size
Platform firmware	520	0
- 2 application tasks - 2 UART Driver tasks - Kernel task - Idle task - OpenComRTOS full MP (_NW, _W, _WT, _A)	230 338 3500	1002, of which - Kernel stack: 100 - Task stack: 4*64 - ISR stack: 64 - Idle Stack: 50 - 568
Total	4138 + 520	1002 + 568

Can be reduced to 1200 bytes code and 200 bytes RAM

Universal packet switching

- Another new architectural concept in OpenComRTOS is the use of “packets”:
  - Used at all levels
  - Replace service calls, system wide
  - Easy to manipulate in datastructs
  - Packet Pools replace memory management
- Some benefits:
  - Safety and security
  - No buffer overflow possible
  - Self-throttling
  - Less code, less copying,

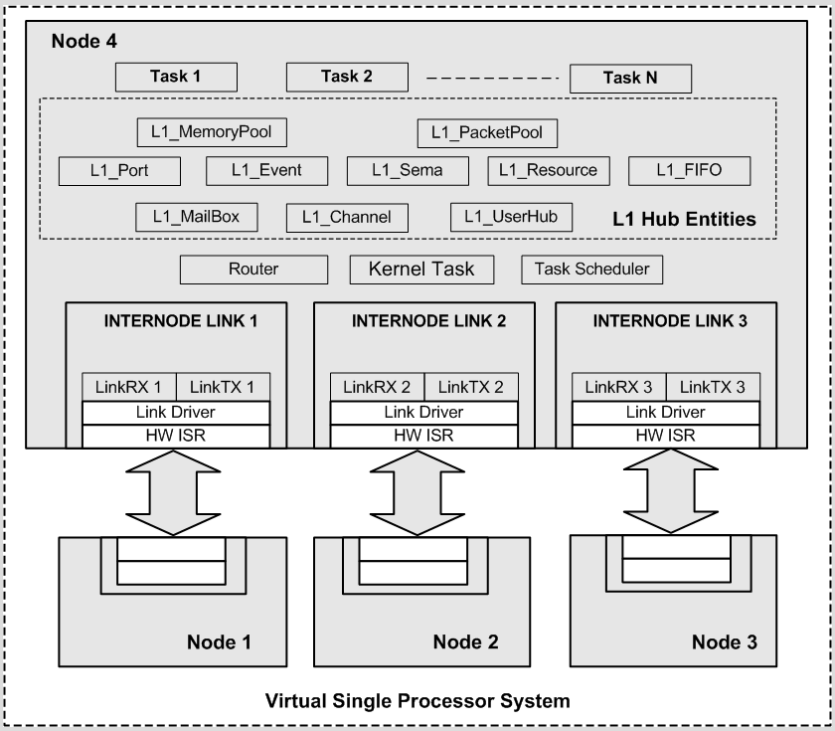
Transparent communication

- Tasks only “communicate” via Hubs
- Real network topology
  - Logical point-to-point links between nodes
  - Node Rx and Tx link driver task for each link end
  - Routing and gateway functionality
  - Works on any medium: shared buses, “links”, “tunneling” through legacy OS nodes using sockets, …
- Link driver tasks
  - Normal OpenComRTOS application task with (TaskInput) Port
  - Driver task type per link type (UART, TCP/IP, …)
  - Not present/visible on the (logical) application level

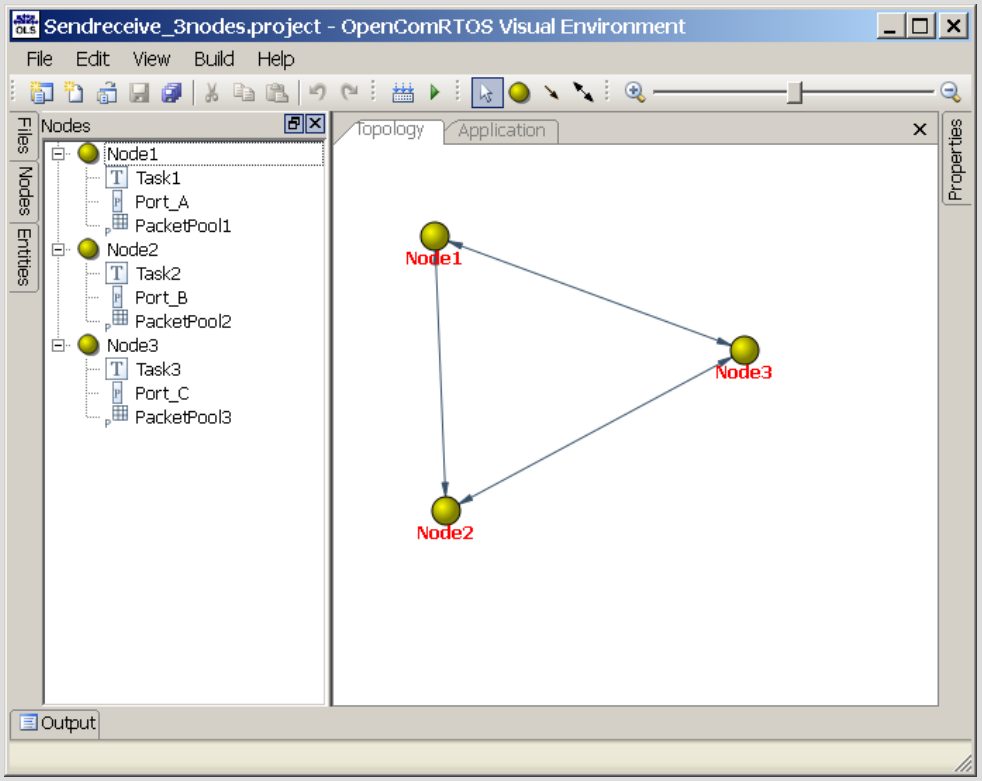
“Link” driver functionality

- Target/link specific communication implementation
  - L0_RxDriverFunction – retrieve packet from “wire”
    - E.g. socket read for Win32, Linux, ..
    - E.g. buffered UART communication for embedded target
  - L0_TxDriverFunction – put packet on “wire”
    - E.g. socket write for Win32, Linux, ..
    - E.g. buffered UART communication for embedded target
  - network <-> host byte ordering functions
  - Normal ISR framework can be used as applicable
- But fully transparent for the application software

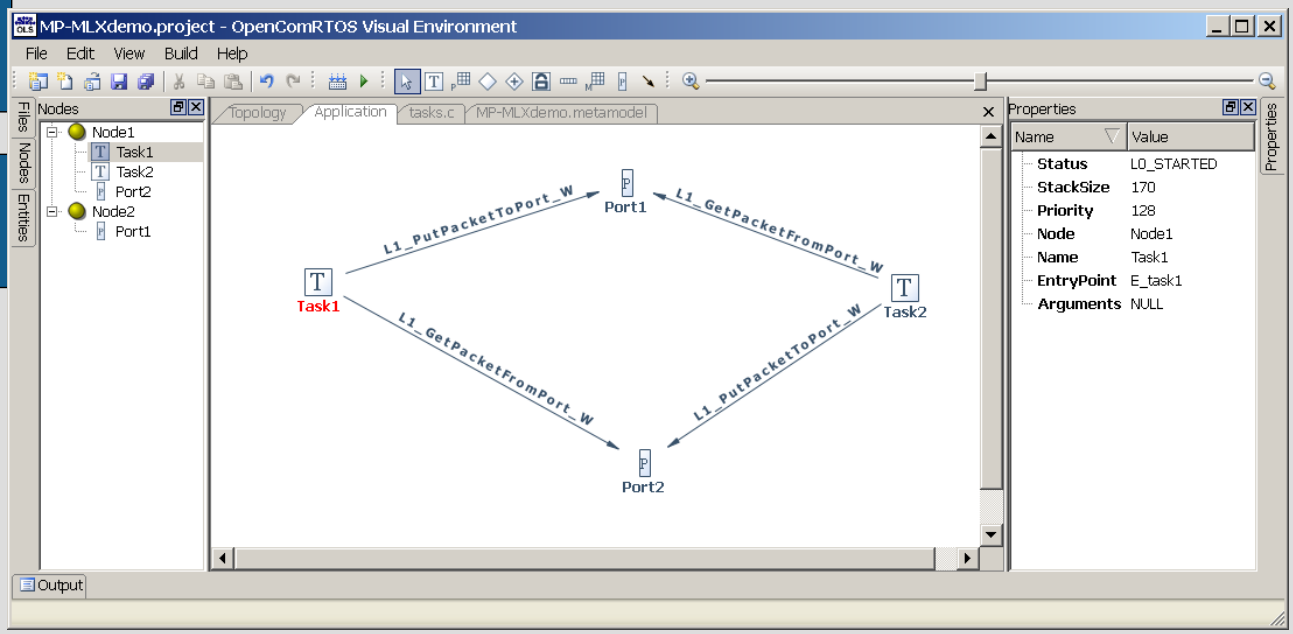
Virtual Single Processor programming model



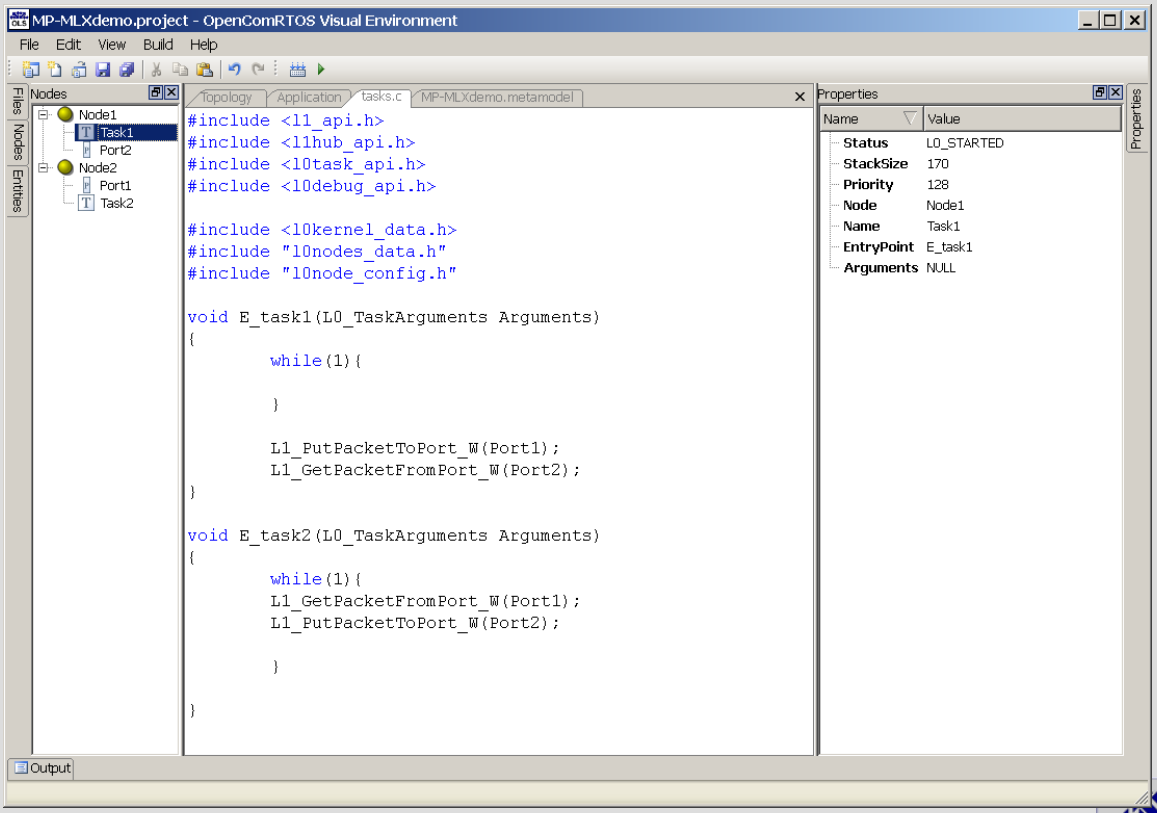
Tool support: Define Topology



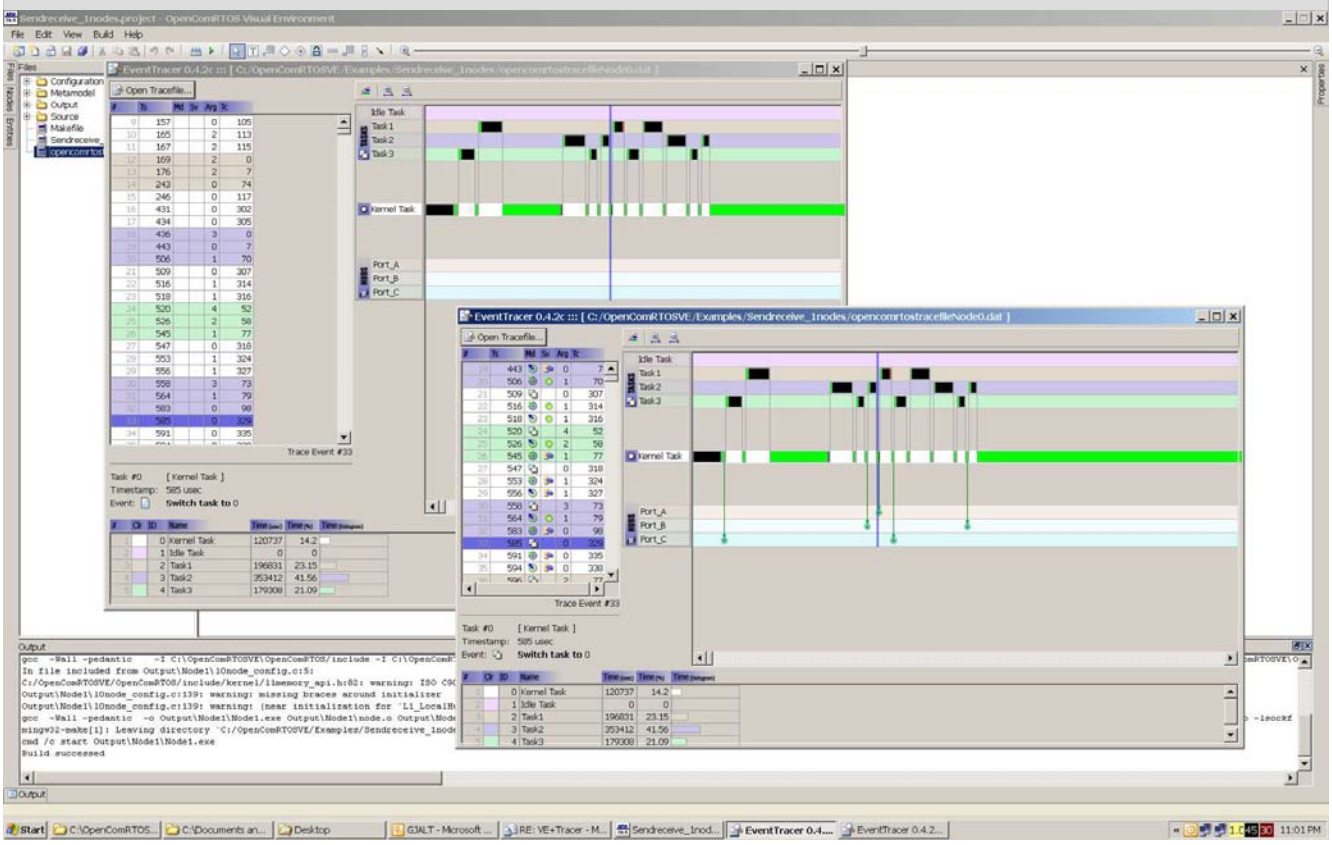
Tool support: Define Application



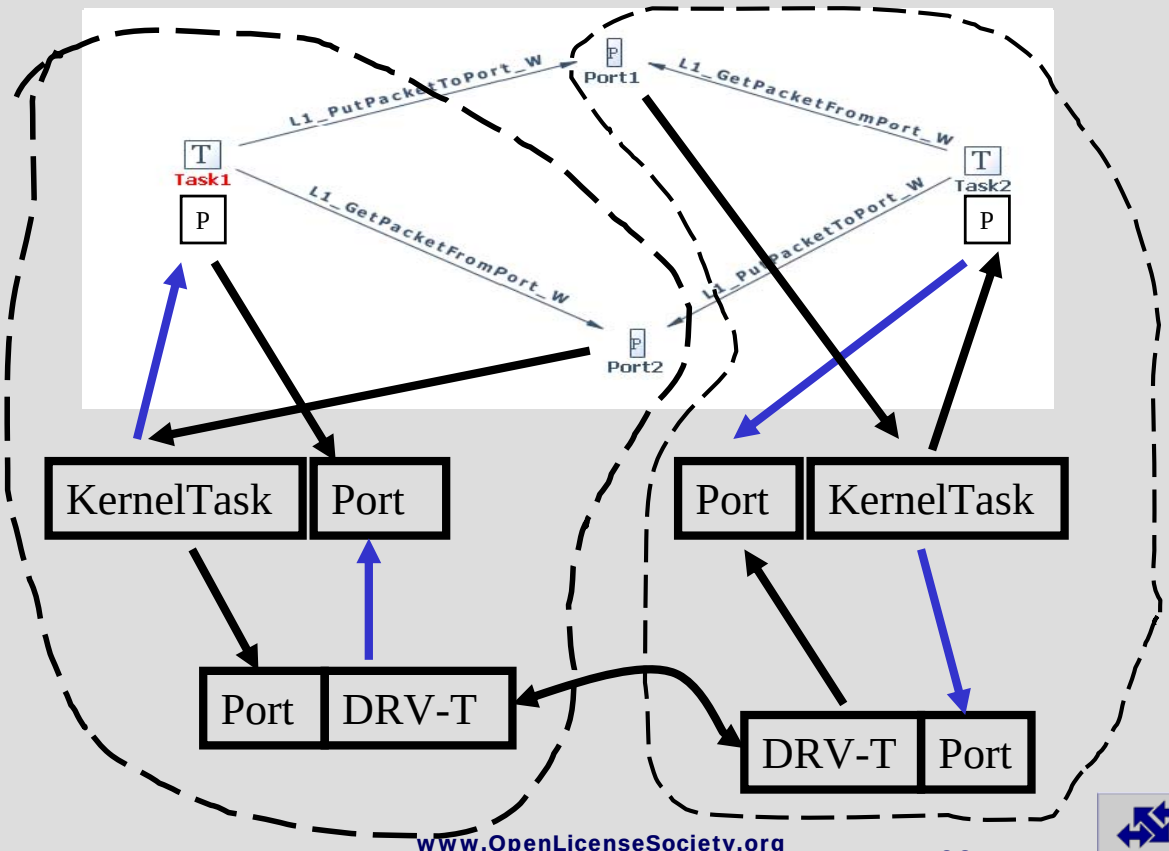
Tool support: C code is generated



Tool support: Run and trace



Under the hood



Heterogenous demo set-up

- Nodes: MLX-16 – UART- AVR –USB - WIN32 - LINUX on virtual host server (via internet)
- Each “node” runs on instance of OpenComRTOS
- Only changes are the node-adresses
- Source code everywhere:

```
....
L1_PutPacketToPort_W (Port1)
...
L1_SignalSemaphore_W (Sema1)
...
```


Conclusions

- OpenComRTOS is breakthrough “RTOS”
  - Network-centric => system communication layer
  - Priority or timer based scheduling => RTOS
  - Formally developed
  - Fully scalable, very safe, very small
  - Better performance
  - Portable & user-extensible
  - => Concurrent programming model
  - => works for any type of “multicore” target
  - ....
- Contact:

  Eric.Verhulst @ OpenLicenseSociety.org

From theoretical concept to products

“If it doesn't work, it must be art.
If it does, it was real engineering”