

# ***A Simple, Lightweight Communications Architecture facilitating SCA application portability***

Charles Linn  
Harris Corporation  
Charles.linn@harris.com

## *Enter stage right... the SCA*



- In early 2000 the Software Communications Architecture (SCA) is introduced to the US military market. This caused a paradigm shift in how radios were designed and structured.
- The SCA formalized waveform / platform separation, component-based design, and waveform portability
  - Military radio SW design “sea change”
  - Once platform APIs standardized, allows waveforms to move between platforms with minimum effort
  - Installable waveforms follow the “PC” model instead of the “Wang word processor model”
- **All these things were good for industry.....**

- Despite advantages, the SCA is often too heavyweight a solution for low-SWAP platforms
  - SCA pretty much assumes multi-process, multiprocessor architectures are required
  - Platform sizes measured in “dozens” of MB, boot times below 10 seconds very hard to achieve
- Desire to keep the good, but devise a lighter weight solution
- Needed – a lighter weight architecture that keeps the advantages of SCA, and fosters application portability to/from SCA frameworks

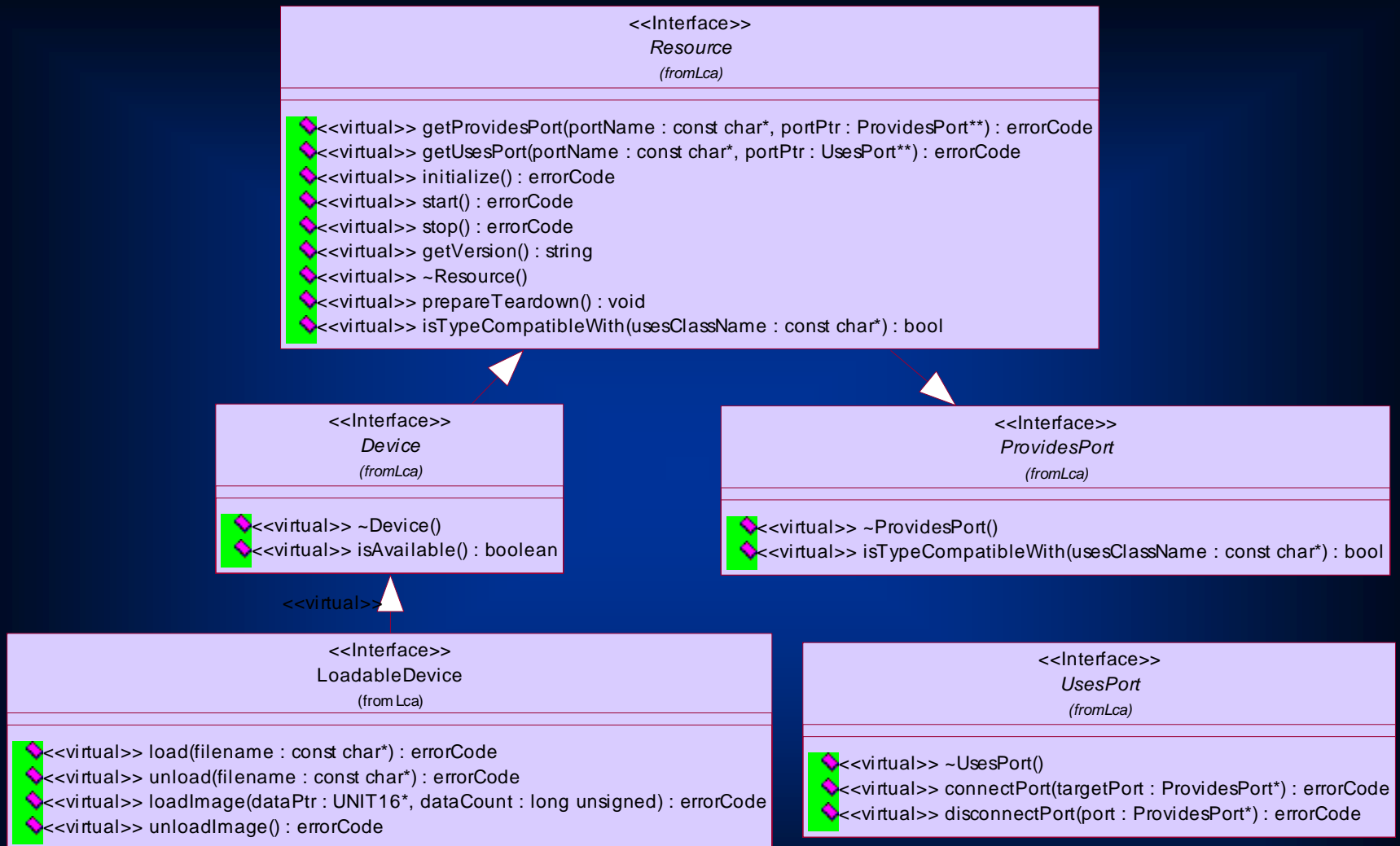
## *Enter stage left... the LCA*



- With this, the Lightweight Communications Architecture (LCA) was born
- Target architecture – small, SWAP-limited platforms employing single-process space, single processor (plus DSP or FPGA) environments
  - This allowed CORBA to be replaced by C++ calls
  - In limited cases multiple processors and address space could be employed by using proxies
- Fostering of waveform portability between SCA and LCA was a primary goal
- Foster developer movement across SCA and LCA projects

- Keep overall SCA structure, and where possible, responsibilities
- Replace CORBA with C++ calls
  - CORBA object references change to C++ pointers
- Standardize on C++ (SCA language porting little used)
- Keep a component model (with ports)
- Keep waveform (Resource) interfaces as similar as practical
- Also create a DevKit to further abstract our SCA / LCA differences
- Specify at interface and postcondition level only
  - Leave implementation choices free

# LCA Component port interfaces

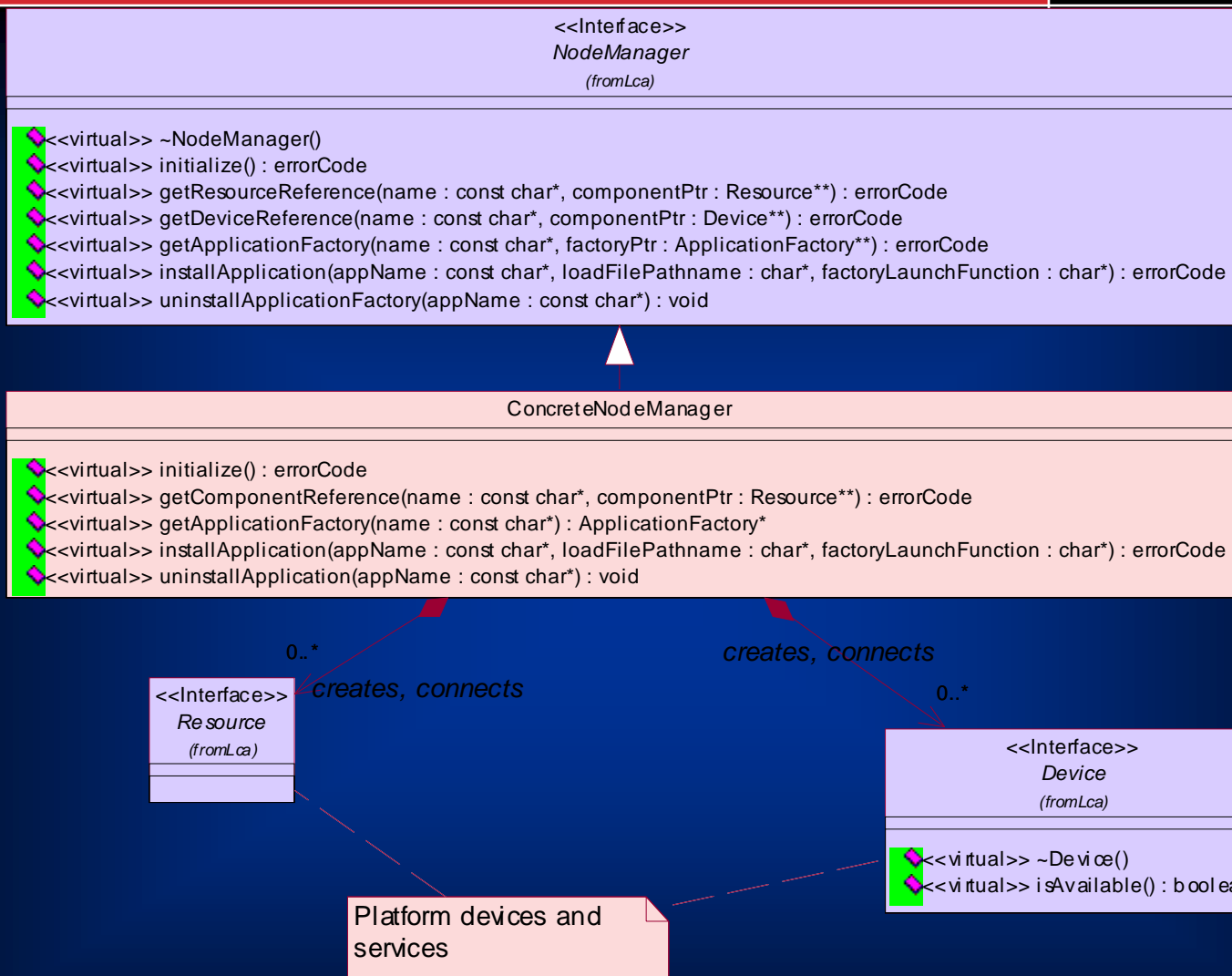


- The SCA Port class was split into ProvidesPort and UsesPort, and explicit type checking was added
  - Maximize strong type checking where possible
  - C++ RTI was not used, as embedded systems
  - A derived port can choose to support the generalized port protocol or not. This supports extension patterns.
- Resource interface maintained, except:
  - Properties were deleted
    - No CORBA any in C++, need lightweight
    - An application assembly controller can add custom configure operations by inheriting from Resource or adding a control ProvidesPort
  - SCA Release operation was changed to a prepareTeardown() (ApplicationFactory does deletion)

- LCA Devices are analogous to SCA devices, but...
  - Don't support *allocateCapacity* or *usageSate*
    - Allocation is done statically in LCA, so no need
  - Don't support *adminState* – since you can't shutdown a node, need disappears
  - The *operationalState* attribute moved to the *isAvailable()* operation
- No *ExecutableDevice* – LCA does not “deploy” components on a Device. Since only one node, they are run by the *NodeManager* itself.
- No *AggregateDevice*
- Services, which are non-HW accessing components launched as part of the platform, use *Resource*.

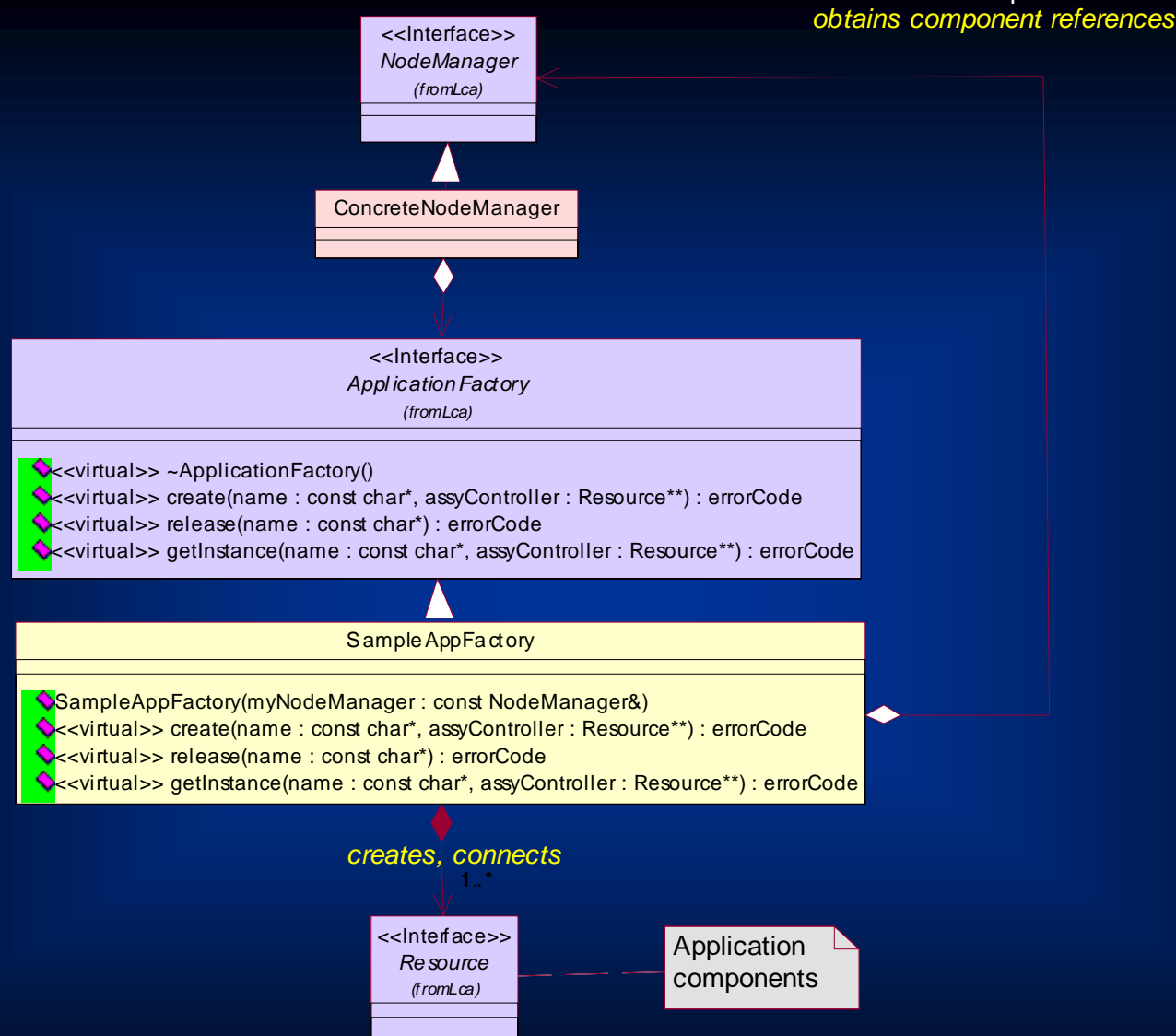


# NodeManager Interface



- The *NodeManager* serves as a combination of a *DeviceManager*, *DomainManager*, and naming service
  - Since LCA scoped to one processing node (plus slaves), multiple *DeviceManagers* merged, and this in turn merged into the *DomainManager*.
  - This merging does not significantly affect applications
- *NodeManager* Responsibilities:
  - Creates all platform *Devices* and services (*Resources*)
  - Creates all *ApplicationFactory* instances
  - Registry of all created resources and app factories
  - Install point for applications (optional)
- *HOW* the platform components and app factories are created is left up to the implementation
  - Hard-coded C++, XML or other based scripts, etc.

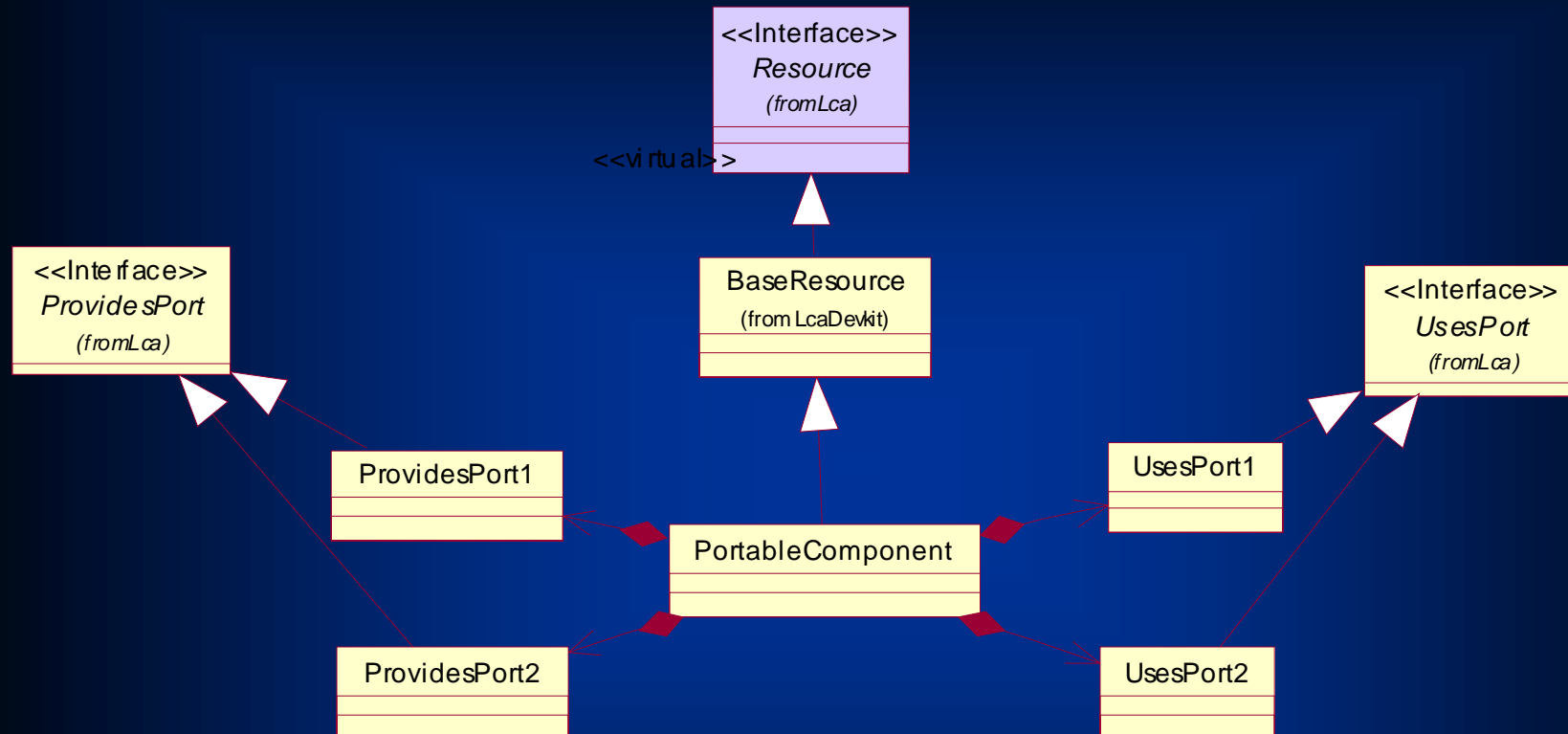
# ApplicationFactory Interface



- *ApplicationFactory* at the interface level is very similar to its SCA cousin, but implementation can be quite different
  - Creates applications
  - Reference counts applications
  - Releases applications (there is no *Application* class) – in this case, only guaranteed behavior is that app. is prepared for teardown and ports disconnected. App. Components may remain instantiated.
- Application is created, and a reference to the *Resource* proxying as the “assembly controller” is returned to caller.
  - Caller should *not* call *prepareTeardown()*, leave this to app factory.
- Implementations can vary:
  - Application creation can be static or dynamic
  - Application creation can be hard-coded, or script based

- As the Resource interface was kept rather similar, designing easily-portable applications is straightforward. To do this, use the following techniques:
  - Use a “DevKit” to provide (and abstract out) common behavior, such as port handling, state handling, and CORBA particulars in SCA
  - Use explicit Port classes (by composition) to convert between (for SCA) C++ and CORBA.
  - This leaves the main component class reasonably free from SCA and CORBA details.
  - Pay attention to threading – try to design application to use 2-way conventions, or use one-way and provide an explicit thread in server. (Even pure SCA has this challenge with co-location)
- Some “CORBAism” is hard to abstract out:
  - Container classes (OctetSequence) – could be abstracted by port classes, at cost of extra copies
  - CORBA object references vs. C++ pointers

# Pattern for LCA ⇔ SCA portability



- LCA is not profound, but earns its value by getting the details right, and hitting a needs “sweet spot”
  - Serves as a common “SCA alternative” framework, such that everyone does not develop their own variants.
  - Keeps a common community with SCA developers
- Application cross-portability can be easily achieved
- Differing implementations can widen range of target platforms.
- The SCA should continue to be used in complex, multiprocessor, or security-centric designs
  - But now we have an alternative...

# Questions?

Contact:

**Chuck Linn**

Harris Corporation

[Charles.Linn@harris.com](mailto:Charles.Linn@harris.com)

