Raytheon

RTI Connext Usage and Architectural Patterns in Radar Product Line Software

George Lafortune

Greg Case

May 17, 2018

Approved for Public Release

Raytheon
Integrated Defense Systems

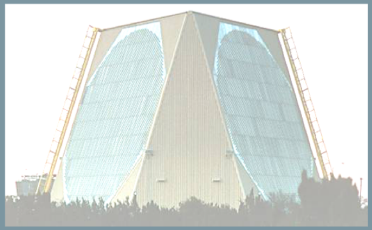
Radar Product Line

Enterprise Air Surveillance Radar

Air and Missile Defense Radar

- Common SW Baseline
- Common Team
- Governance

And others...



Radar Product Line Numbers

- Millions - source lines of code
- 100+ - typical number of servers per radar
- Hundreds - approximate number of DDS topics
- 64 MB /s - approximate throughput required over our more stressing DDS connections
- 26 GB - approximate max size of one of our larger send queues for a reliable Data Writer

RTI Connext – What We Like and Use Today

- Comprehensive documentation
- Responsive and high quality tech support
- Tools (Admin Console, Monitor, DDS Spy, DDS Ping)
- Developer license model
- Prototyper
  - Extensive use of Prototyper and Lua for test drivers and emulation of system components

RTI Connext – What We Want To Use In The Future

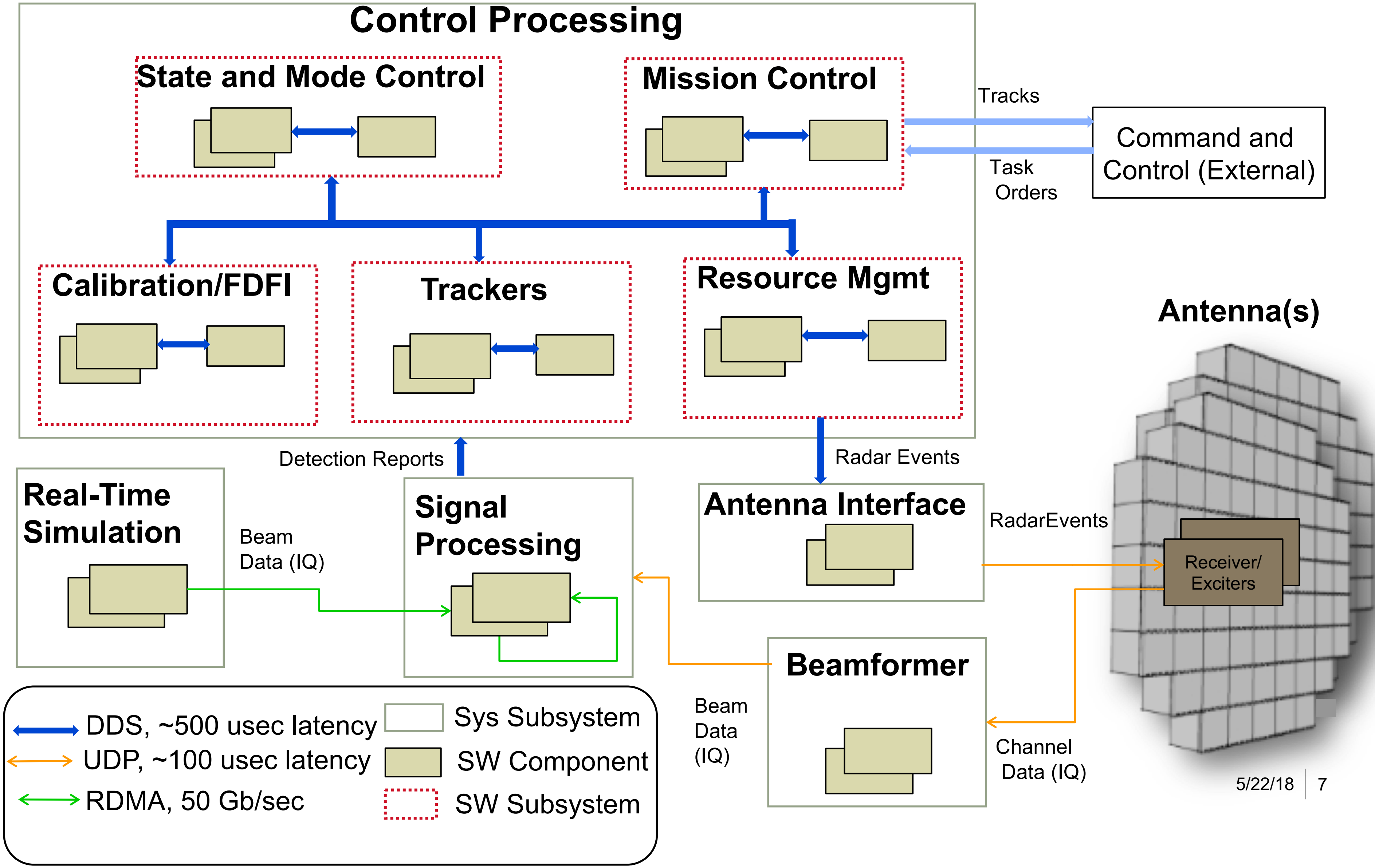
- Extensible Types
  - Maintain backwards compatibility

RTI Connext – What Could Be Improved

Raytheon
Integrated Defense Systems

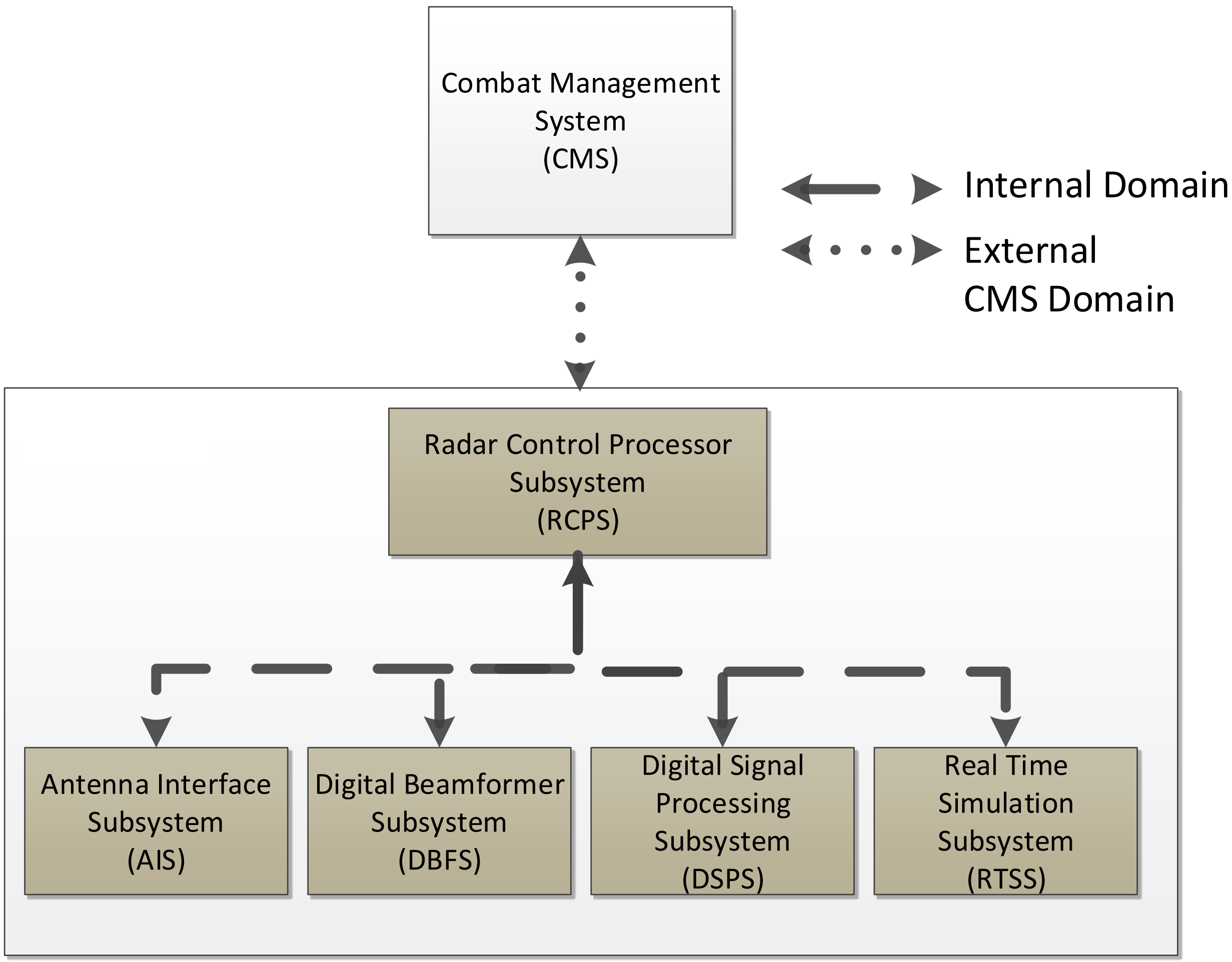
- Options for optimizing serialization performance
- Documentation organization
- Infiniband support
- Application error notifications

Simplified Architecture of a Notional Radar System



Use of DDS Domains

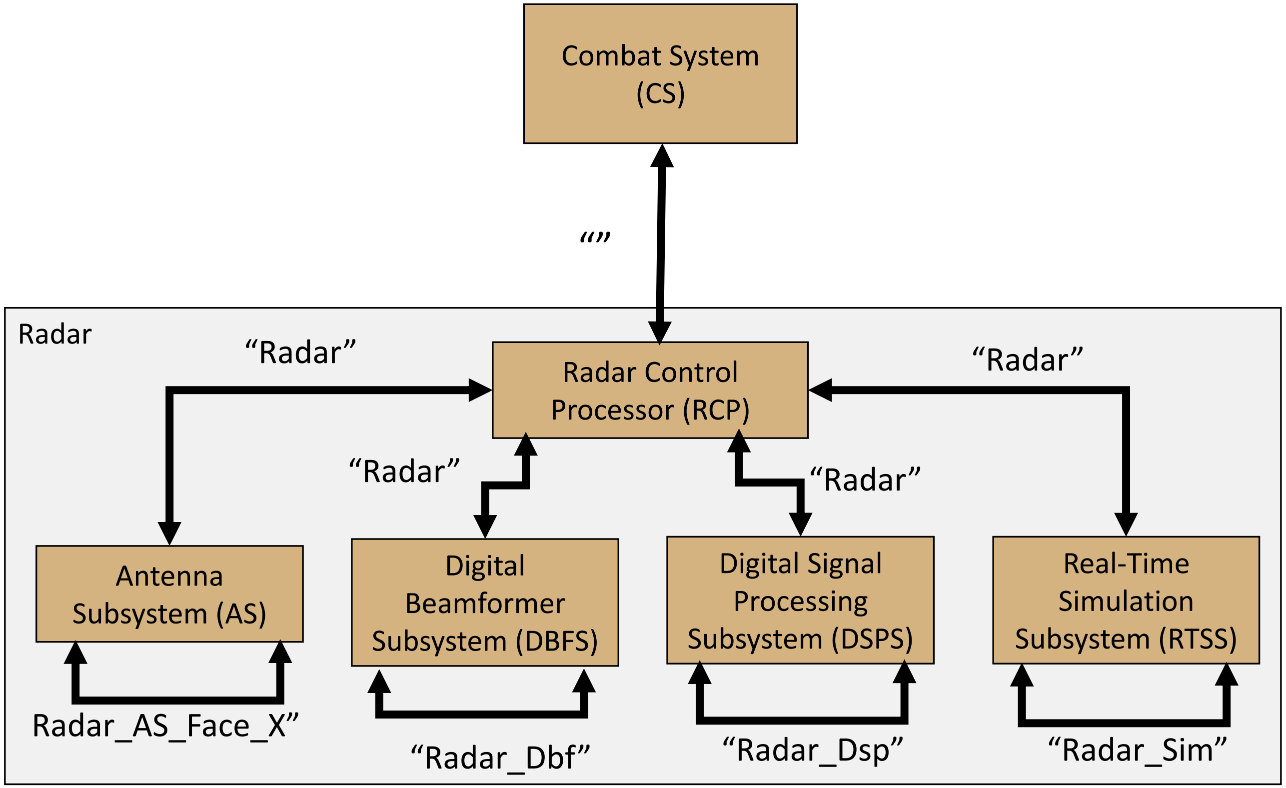
Raytheon
Integrated Defense Systems



- Single domain for the Radar internal Communication
- Separate domains for external interfaces

Use of DDS Partitions

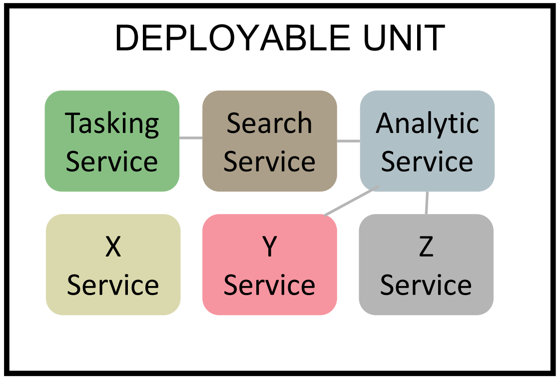
Raytheon
Integrated Defense Systems



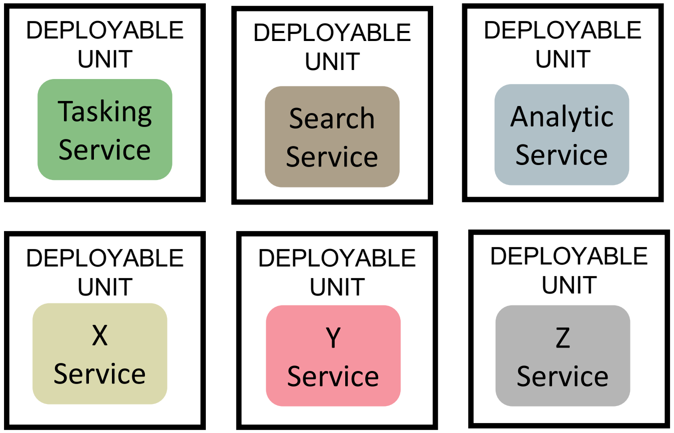
- Subsystems communicate internally on their own partitions
  - Subsystems can be developed by third parties
  - Avoids chances of topic name conflicts
- Have also proven useful to configure an input source dynamically
  - E.g. Simulated Hardware vs. Live Hardware
- There is also a fault tolerance application (discussed later in this briefing)

Microservices

- What is a Microservice?
  - A loosely coupled, independently deployable, fine-grained service
  - Separately compilable (e.g. .exe, .so, .a)
  - A well-defined API, typically HTTP in the business domain but can be any protocol
- What is a Microservice Architectural Style?
  - An approach to building a software application as a suite of fined grained services
- A Microservice Architecture is the opposite of a Monolithic Architecture
  - A Monolithic architecture groups all the functionality of the system into a small number of large executables, often just 1
- A Microservice Architecture supports DevOps
  - Independently testable and deployable fine grained units



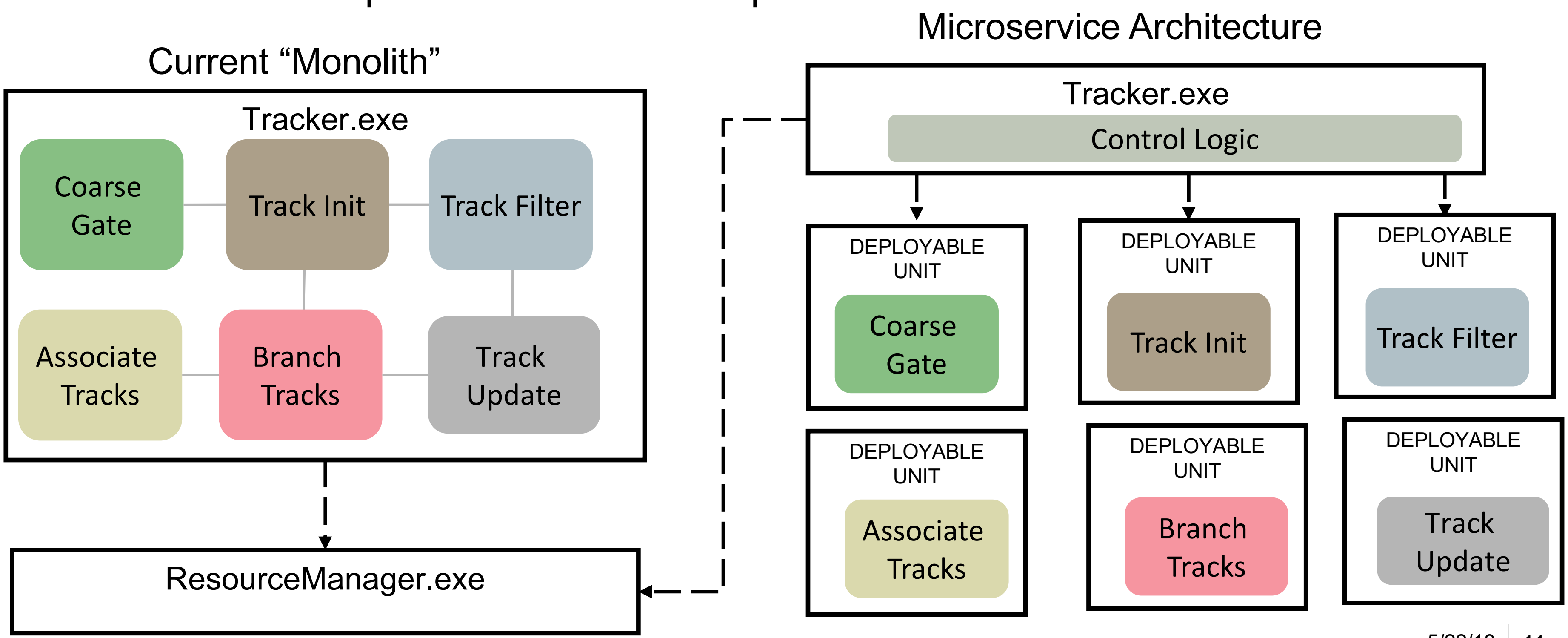
Monolith



Microservices

Product Line Architecture Style

- Architectural style is somewhere between a Monolith and a very good Microservices Architecture
- Some of our library based services are very good examples of Microservices
  - E.g. Frequency Selection
- Some of our executables could be more optimally decomposed into Microservices and be more independent of other components

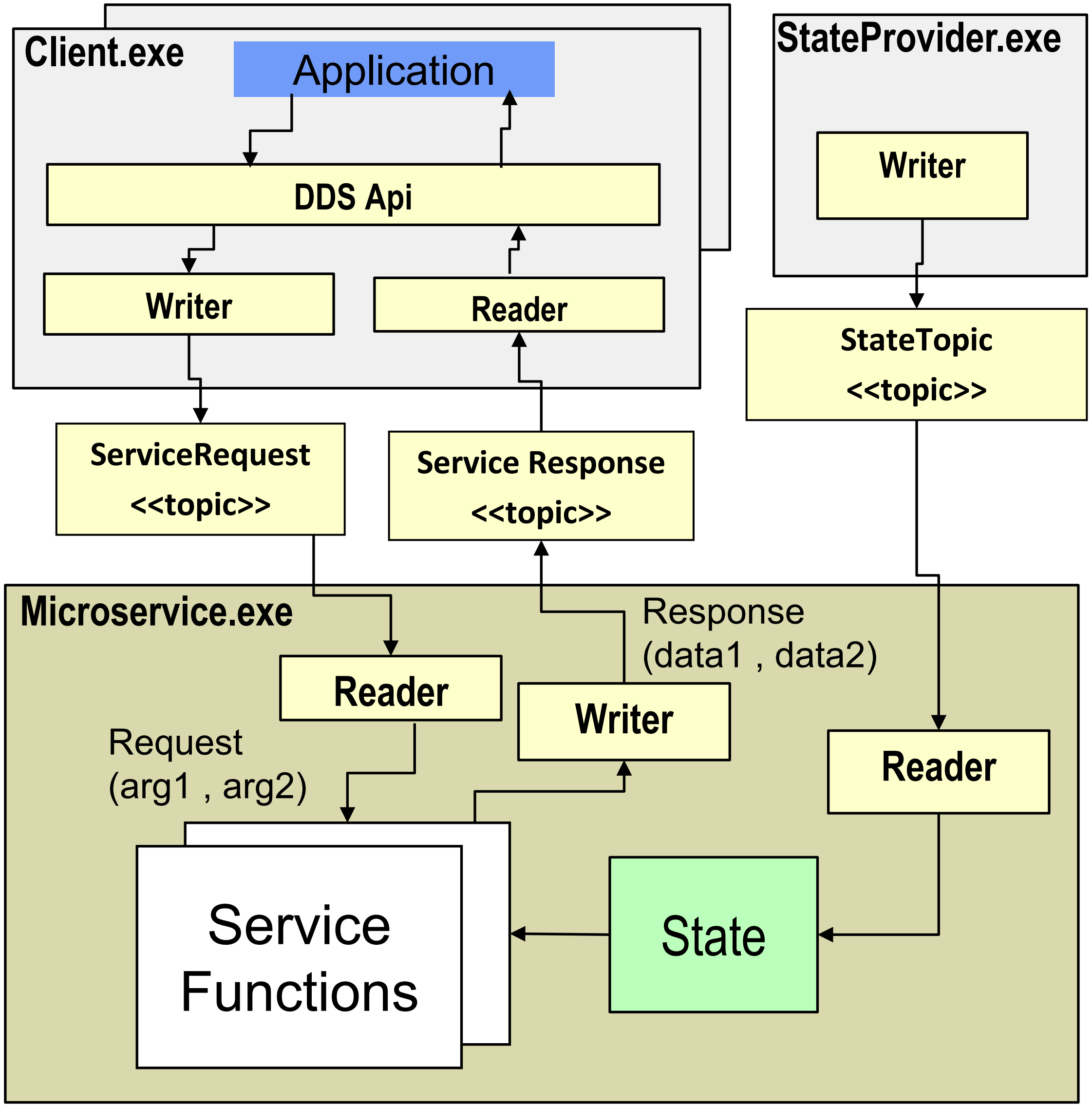


The SBPL "Distributed Microservice" Architectural Pattern

- Applicability
  - Multiple clients within the system require a common lightweight service that internally utilizes a globally consistent set of state data
- Design Forces
  - Service calls must have low latency
  - Clients are distributed
  - Internal data sets must be globally consistent
  - Easy to incorporate in new client applications
  - Support for safety critical processing
- Participants
  - State Provider: Component that provides data to the Microservice
  - Client: User of the Microservice
  - Microservice: Implements service function, returns results to clients

Raytheon
Integrated Defense Systems

Microservice Design – V1



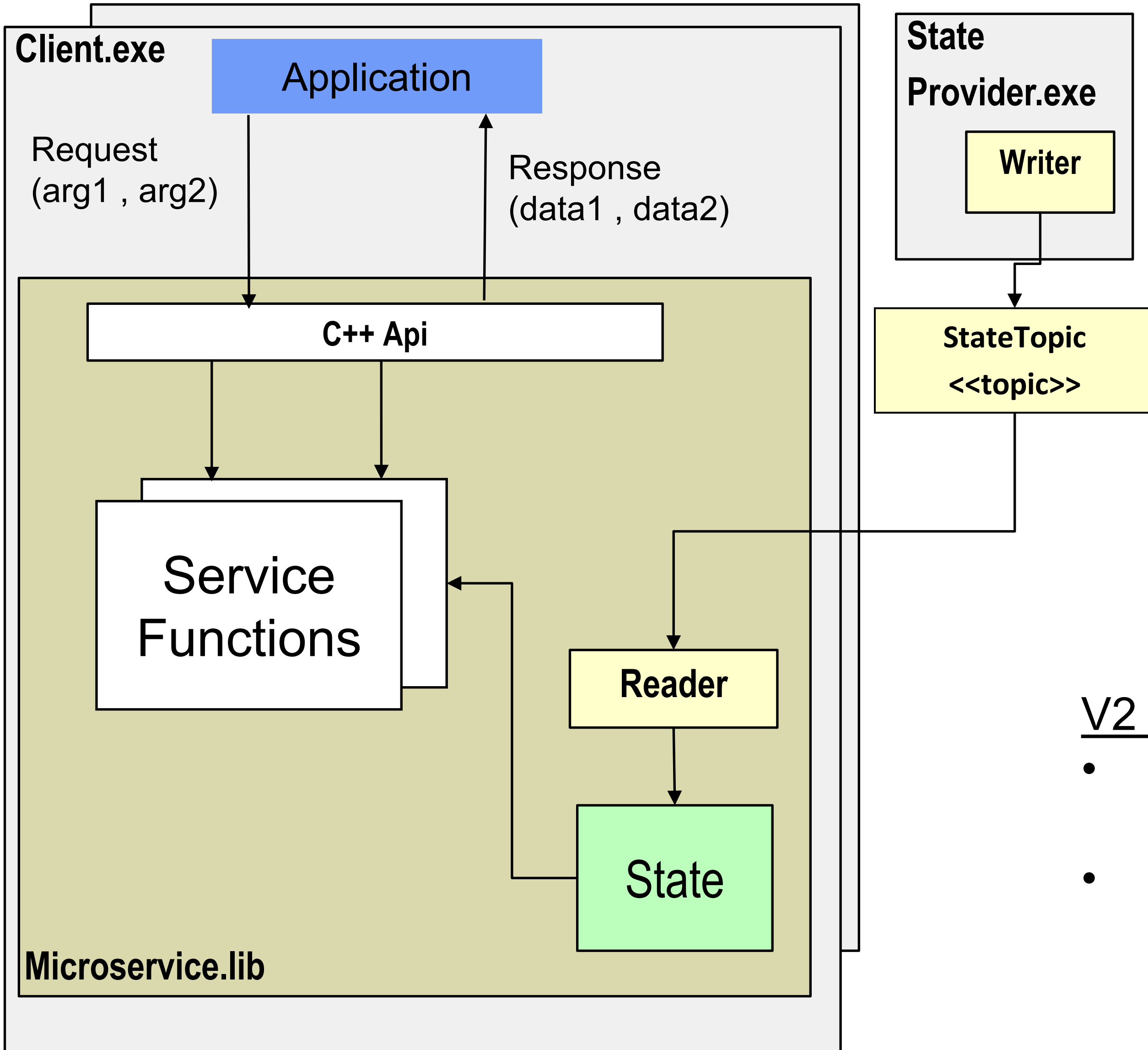
<u>V1 Solution:</u>

- Microservice as a single component instance in the system
- Exposed DDS API
- State provider publishes data to service

<u>V1 Limitations:</u>

- Distributed service latency too high
- Every Client re-implements DDS plumbing

Microservice Design – V2



<u>V2 Solution:</u>

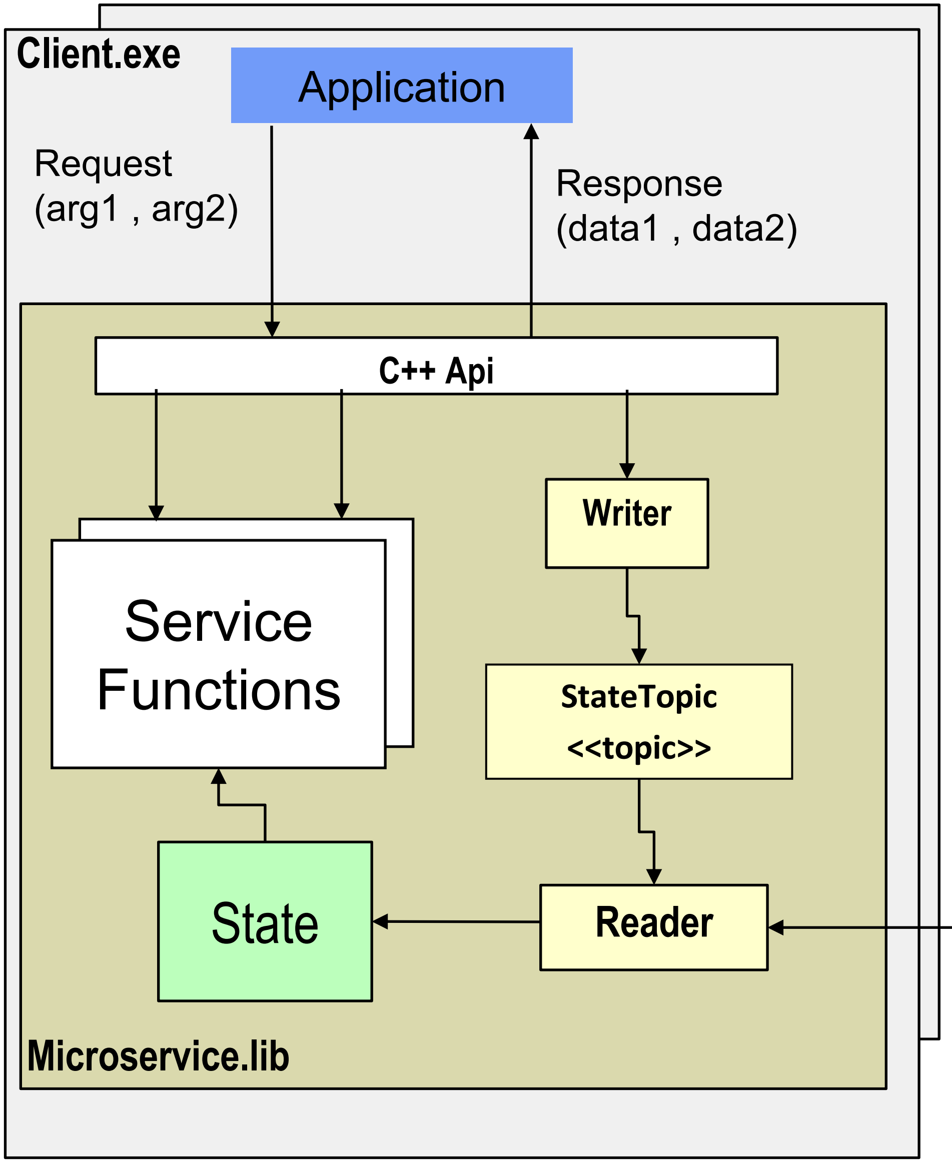
- Implement microservice as a library
- State Provider publishes to all instances
- Provide a C++ API to the microservice
- All DDS code encapsulated within the microservice

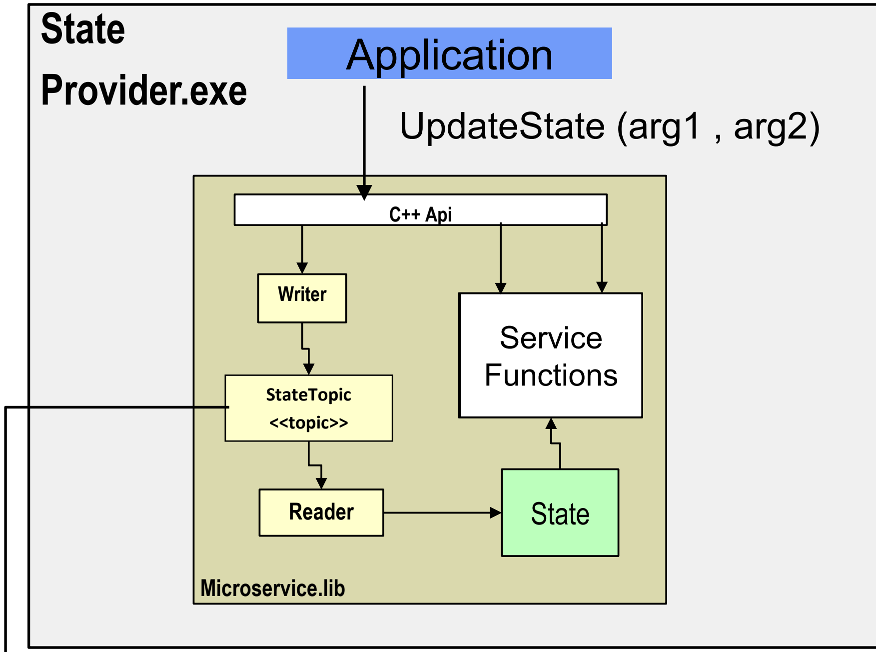
<u>V2 Limitations:</u>

- State Provider must implement DDS plumbing
- Would like ack/nack for safety critical state messages

Microservice Design – V3

Raytheon
Integrated Defense Systems



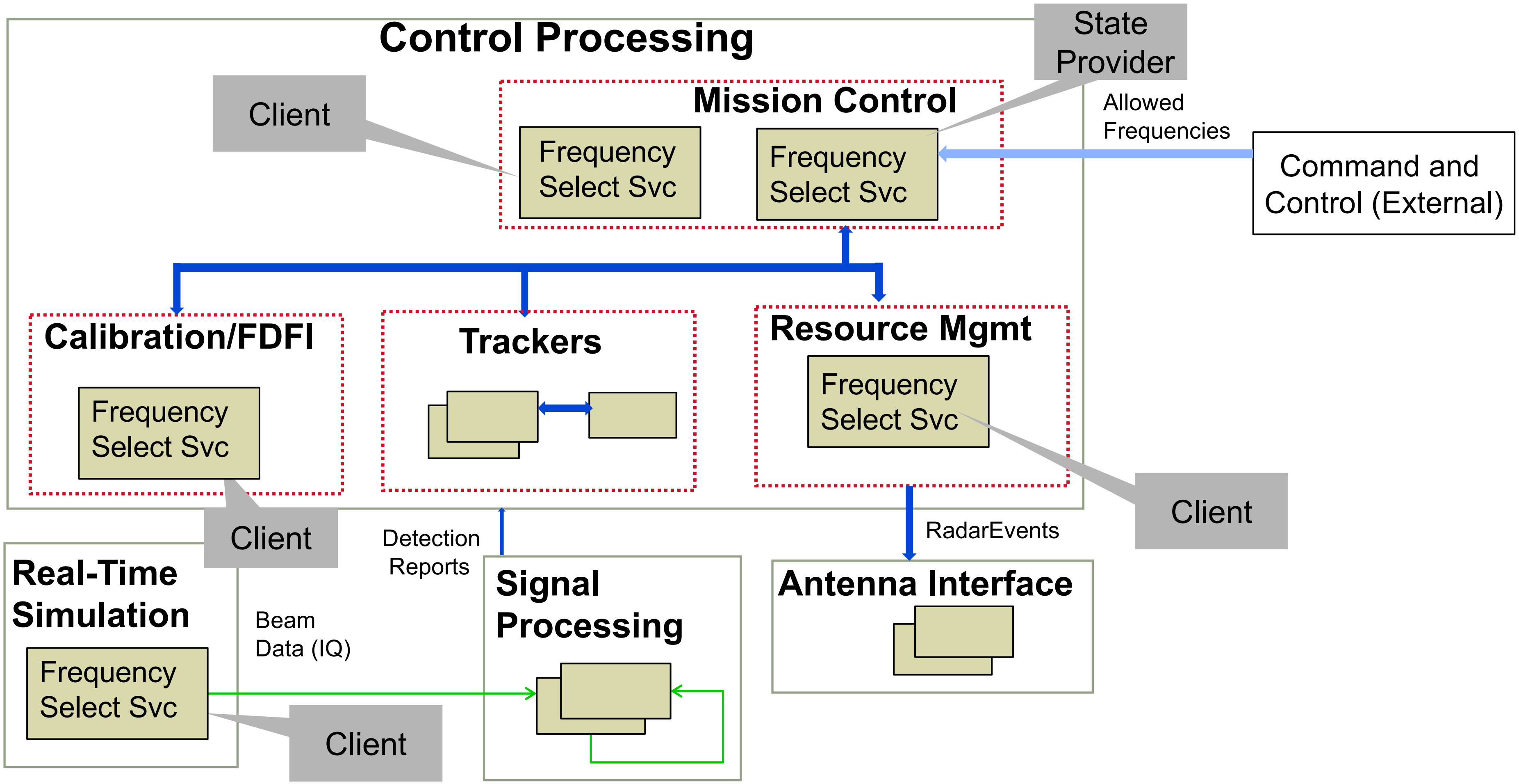


<u>V3 Solution:</u>

- Encapsulate state publishing DDS code in microservice with associated C++ API
- State Provider instantiates service using C++ API
- Add application ack/nack for safety critical usages

Example Radar Microservice – Frequency Selection

Raytheon
Integrated Defense Systems



Microservices easily instantiated where needed in the system

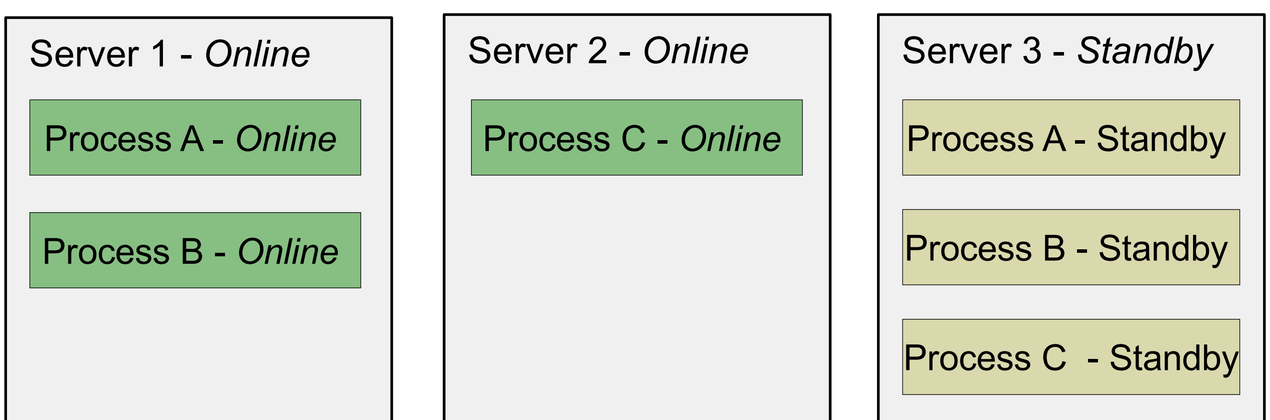
Example Microservices in Raytheon's Radar Product Line

Raytheon
Integrated Defense Systems

Service	Description	Data Set
Coordinate Transform	Provides conversions among different coordinate frames	Ship motion data
Frequency Selection	Chooses RF frequency based on client policy selection	• Allowed/disallowed frequencies • Jammed/clear frequencies
Power Constraint	Provides beam correlation checks against defined set of constrained power sectors	Power sector definitions
Clutter Map	Provides clutter information for given location	Clutter Map Cells
Data Recording Service	Provides services for real-time data recording	Allowed/disallowed collection points

Fault Tolerance

- What is Fault Tolerance?
  - The capability for a system to continue to operate with little or no degradation in the presence of component or hardware faults
  - For Raytheon's Radar Product Line Software, the major driving requirement is recovering from server or network failures
- Design forces for Fault Tolerance in software
  - Maintain consistent state
  - Minimize interruption of service
  - Easy to make new or legacy software components fault-tolerant
- Nominal Architecture:



One or more Standby servers provide redundancy for Online servers (e.g. N+1 model)

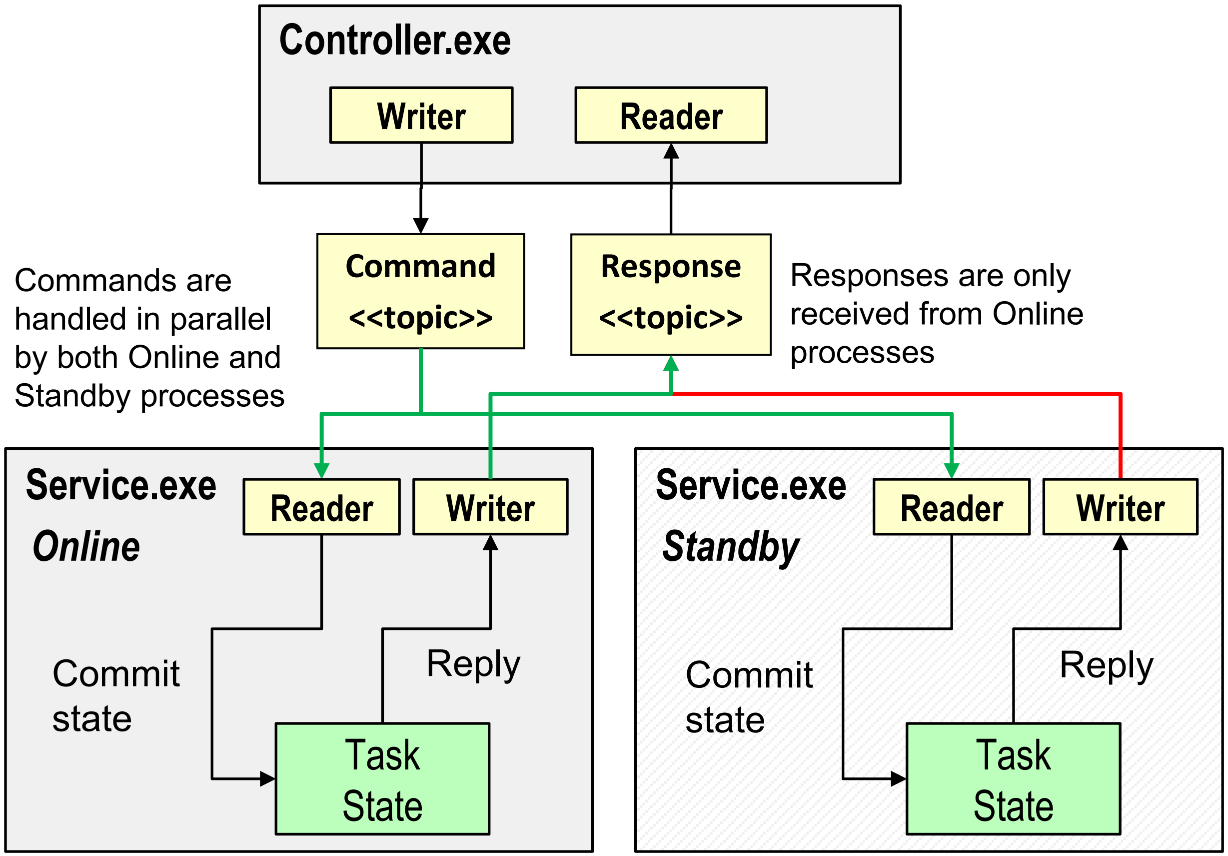
Common Fault Tolerance Approaches

Raytheon
Integrated Defense Systems

Approach	Description	Latency	Complexity
Cold Standby	• Restart on failure • Periodic checkpoints to disk • State loaded on restart	High	• Least complex • Non-mission critical systems
Warm Standby w/ State Checkpoint to Disk	• Alive but inactive process • Periodic checkpoints to disk • State loaded on takeover	Medium	• Medium complexity
Warm Standby w/ Real-time Checkpointing	• Active process • Real-time checkpointing	Low	• High complexity
Hot Standby / Shadow Processing	• Active process • Requests / data handled in parallel	Lowest	• Highest complexity • Best for stateless processing

Fault Tolerance Challenge 1

Hot Standby Risks Inconsistent State

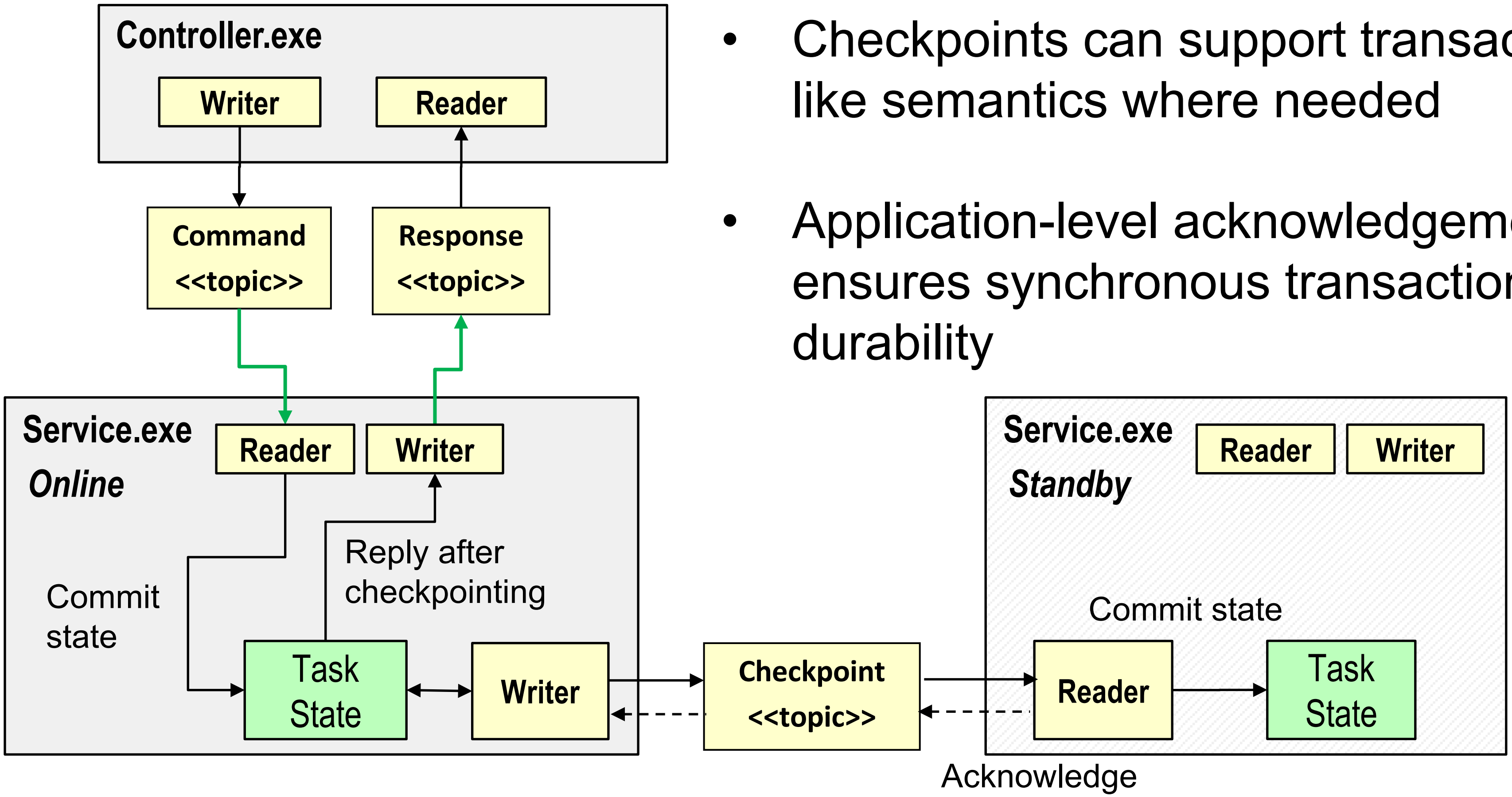


Issues:

- Small variations in conditions (time, order, race conditions, etc.) can cause standby to get a different answer
- After fail-over, status may no longer be consistent with original request

Fault Tolerance Solution 1

Warm Standby With Realtime Checkpointing



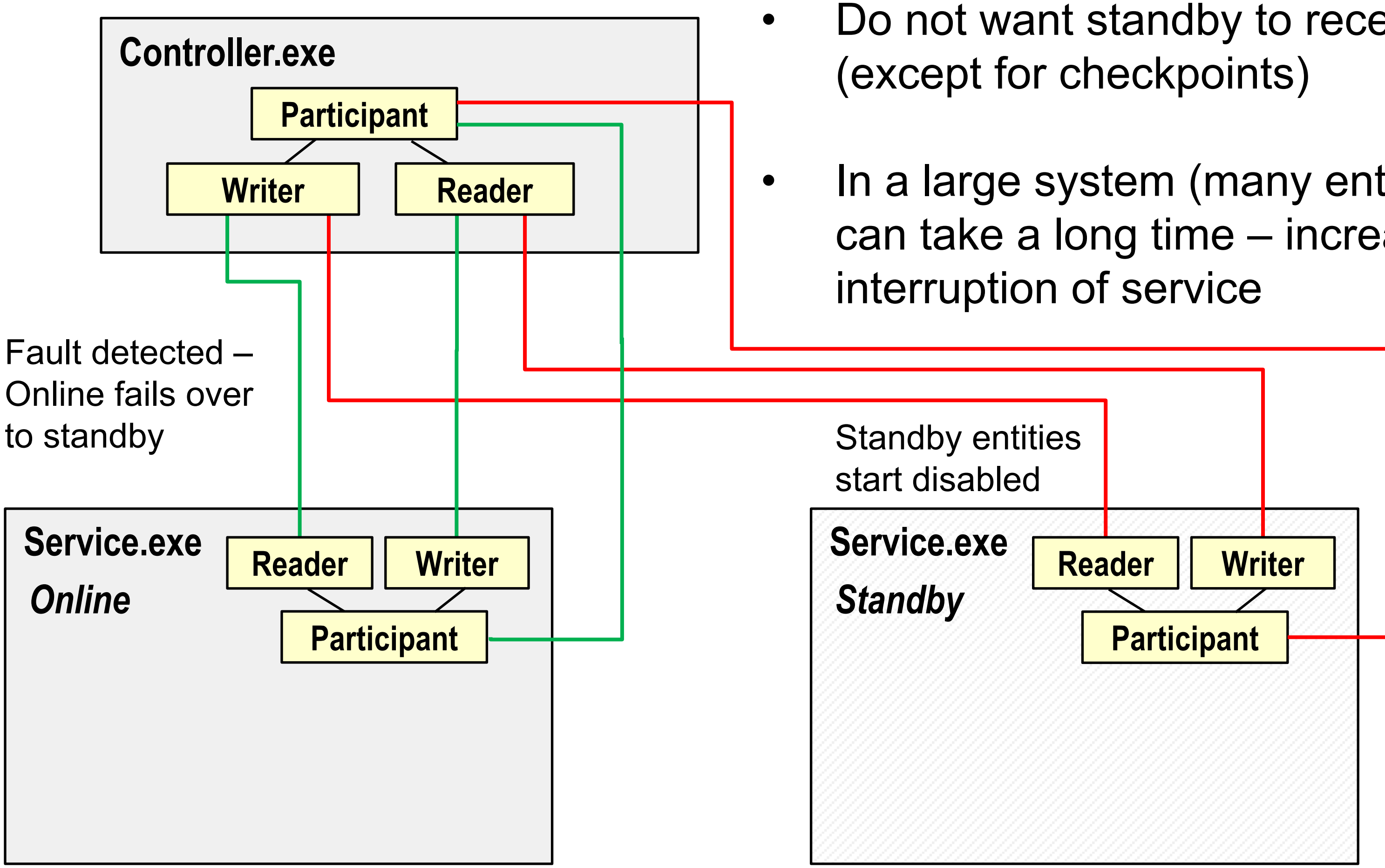
- Checkpoints can support transaction-like semantics where needed
- Application-level acknowledgement ensures synchronous transaction durability

Fault Tolerance Challenge 2

Minimize Interruption of Service

Issues:

- Do not want standby to receive data (except for checkpoints)
- In a large system (many entities), Discovery can take a long time – increasing interruption of service

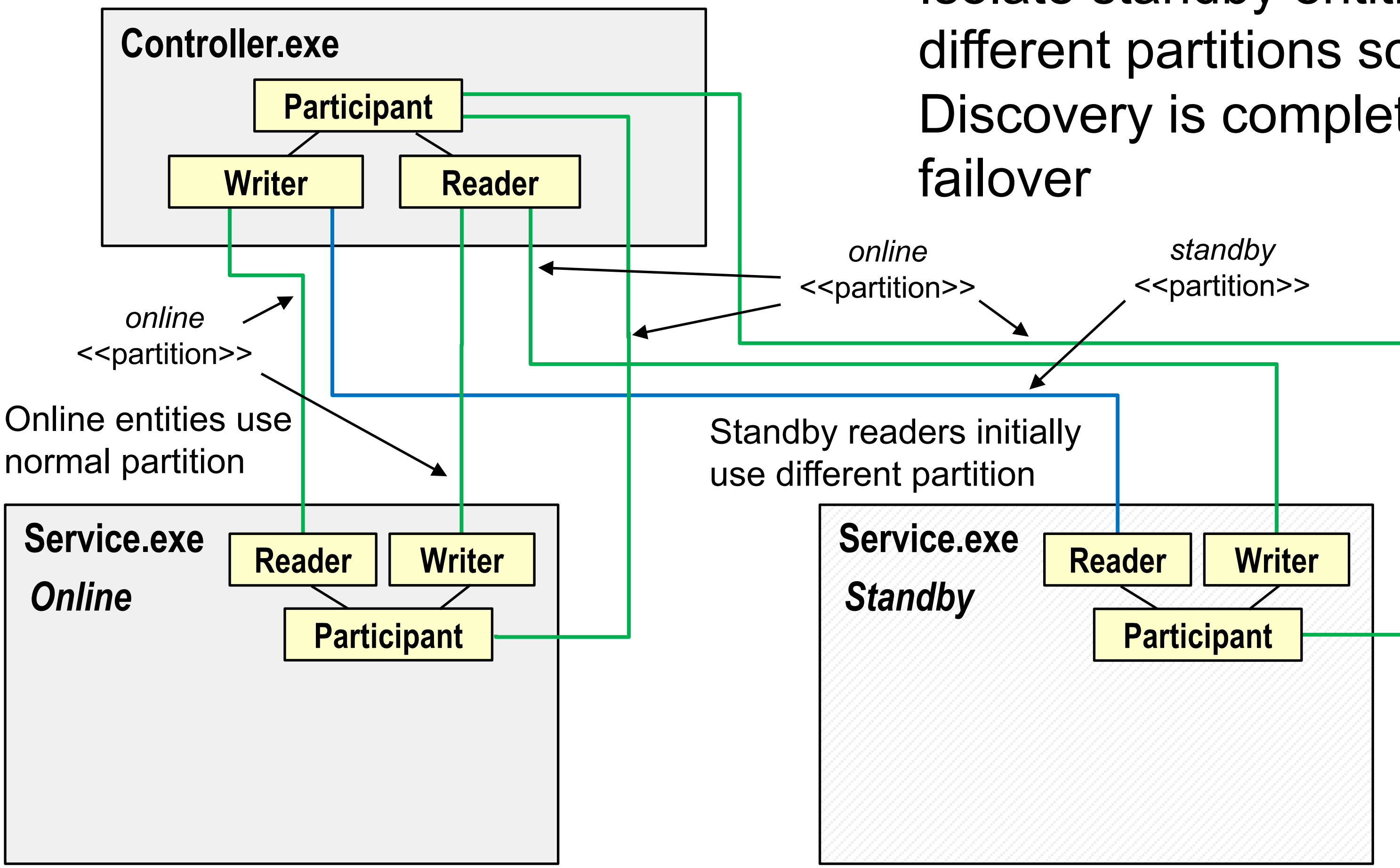


Service not restored until standby entities have discovered endpoints

Fault Tolerance Solution 2

Use Partitions to Reduce Recovery Time

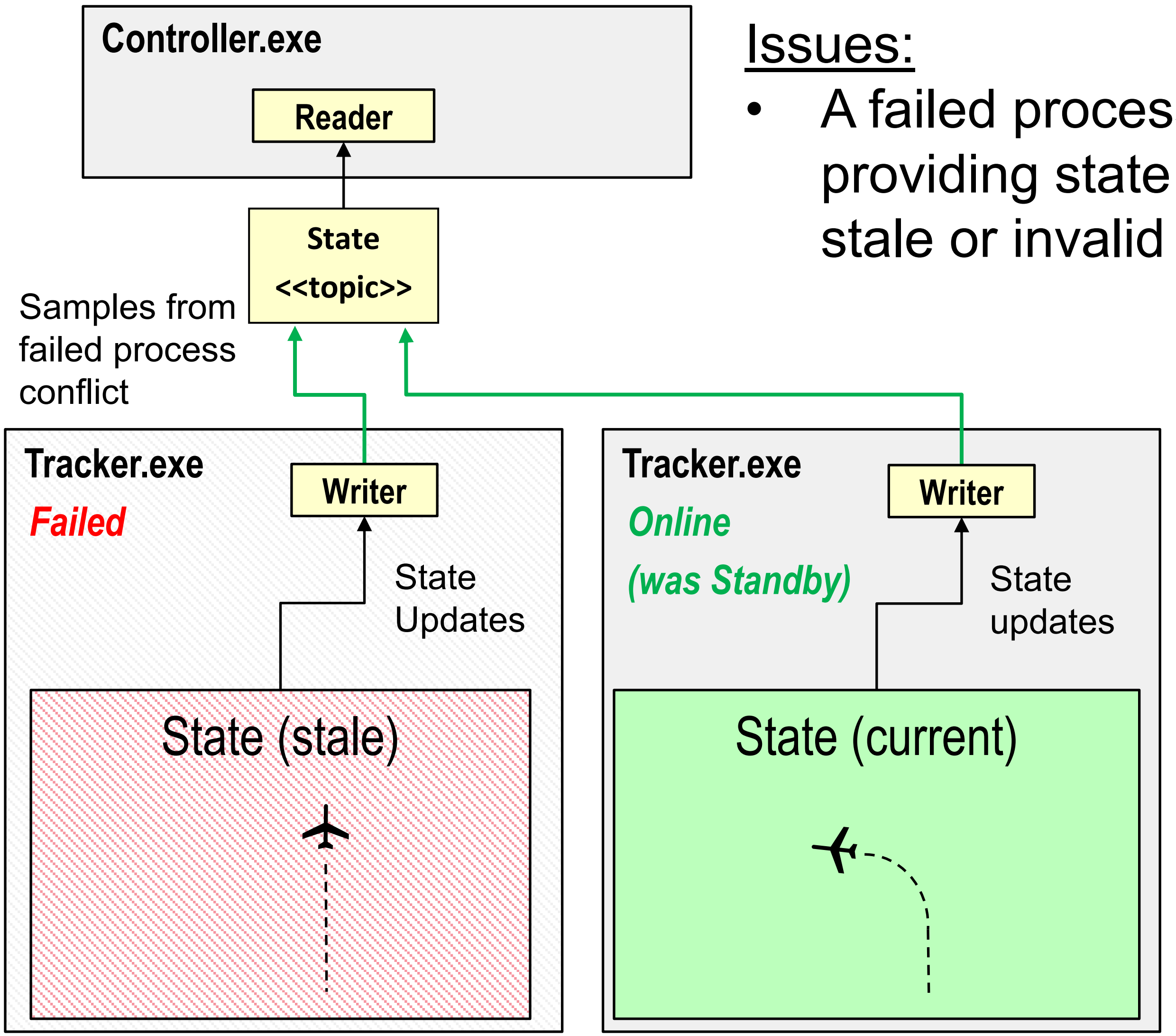
- Isolate standby entities via different partitions so that Discovery is complete prior to failover



On failover, service restored once partition QoS change is propagated to peers

Fault Tolerance Challenge 3

Need to Fence Off Failed Nodes

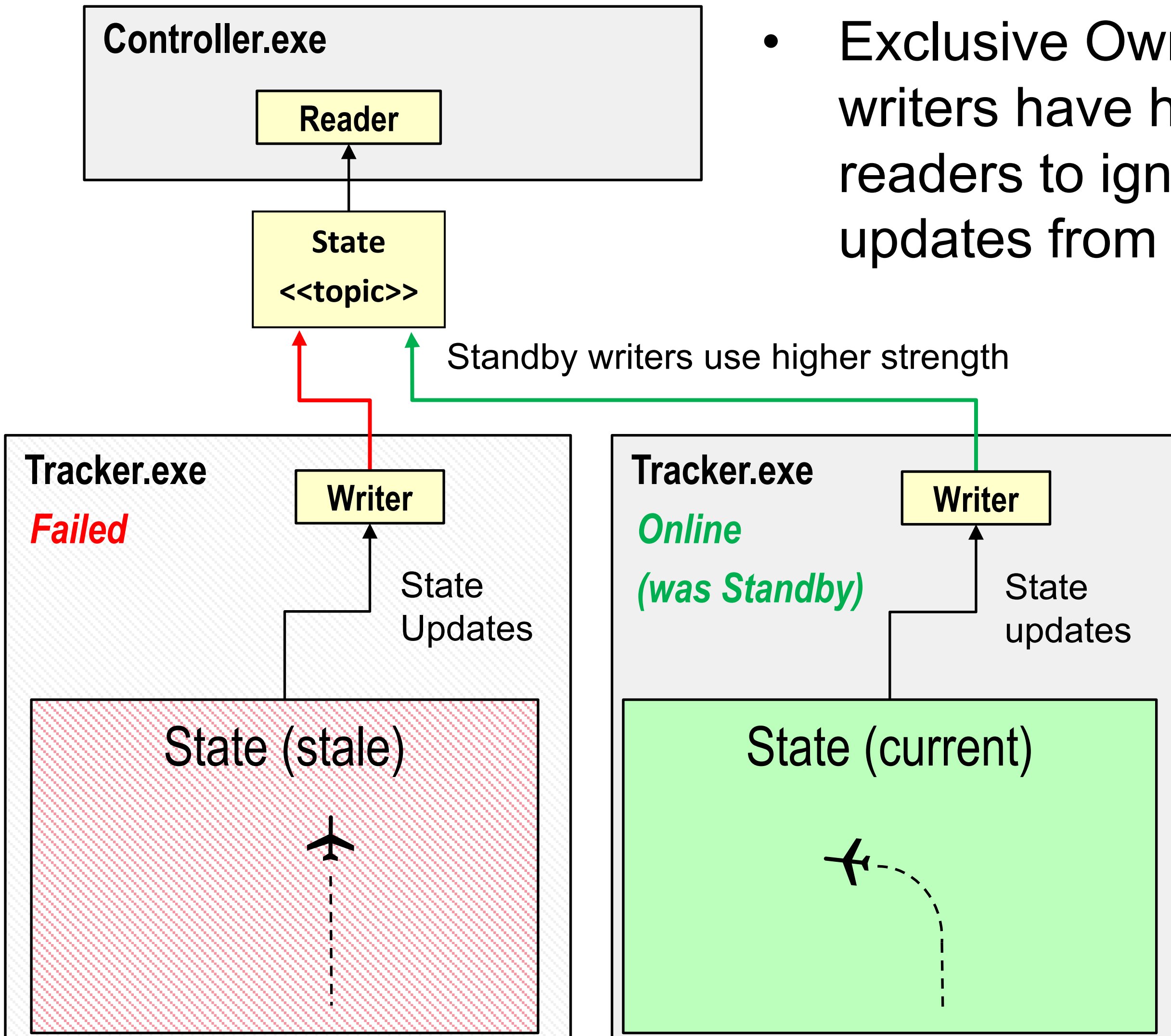


Issues:

- A failed process may still be providing state updates based on stale or invalid state during teardown
- If Ownership is shared, readers may receive conflicting samples
- If online has higher Strength, readers may receive only samples from failed process

Fault Tolerance Solution 3

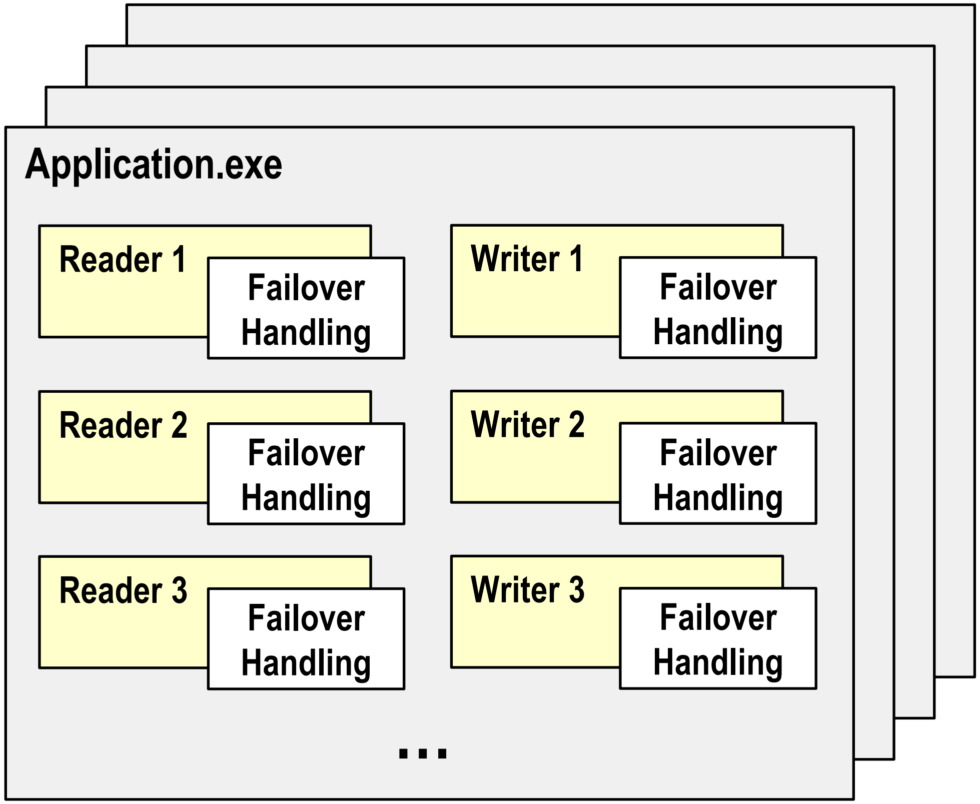
Standby Has Higher Ownership Strength



- Exclusive Ownership where standby writers have higher Strength allows readers to ignore stale / invalid updates from failed processes
- Ownership is transferred as soon as standby process begins publishing each instance

Fault Tolerance Challenge 4

Many Entities Must Be Failover-Aware

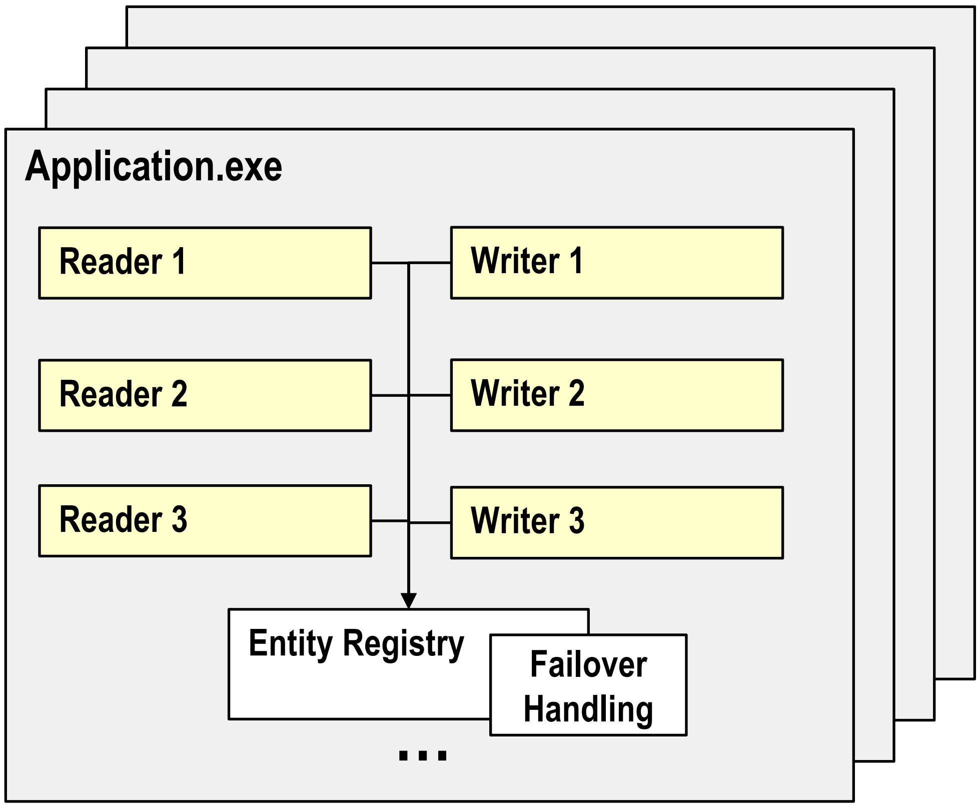


Issues:

- Many entities need to be failover-aware
- Need to 'touch' many parts of the code – can be costly even when common helpers are available

Fault Tolerance Solution 4

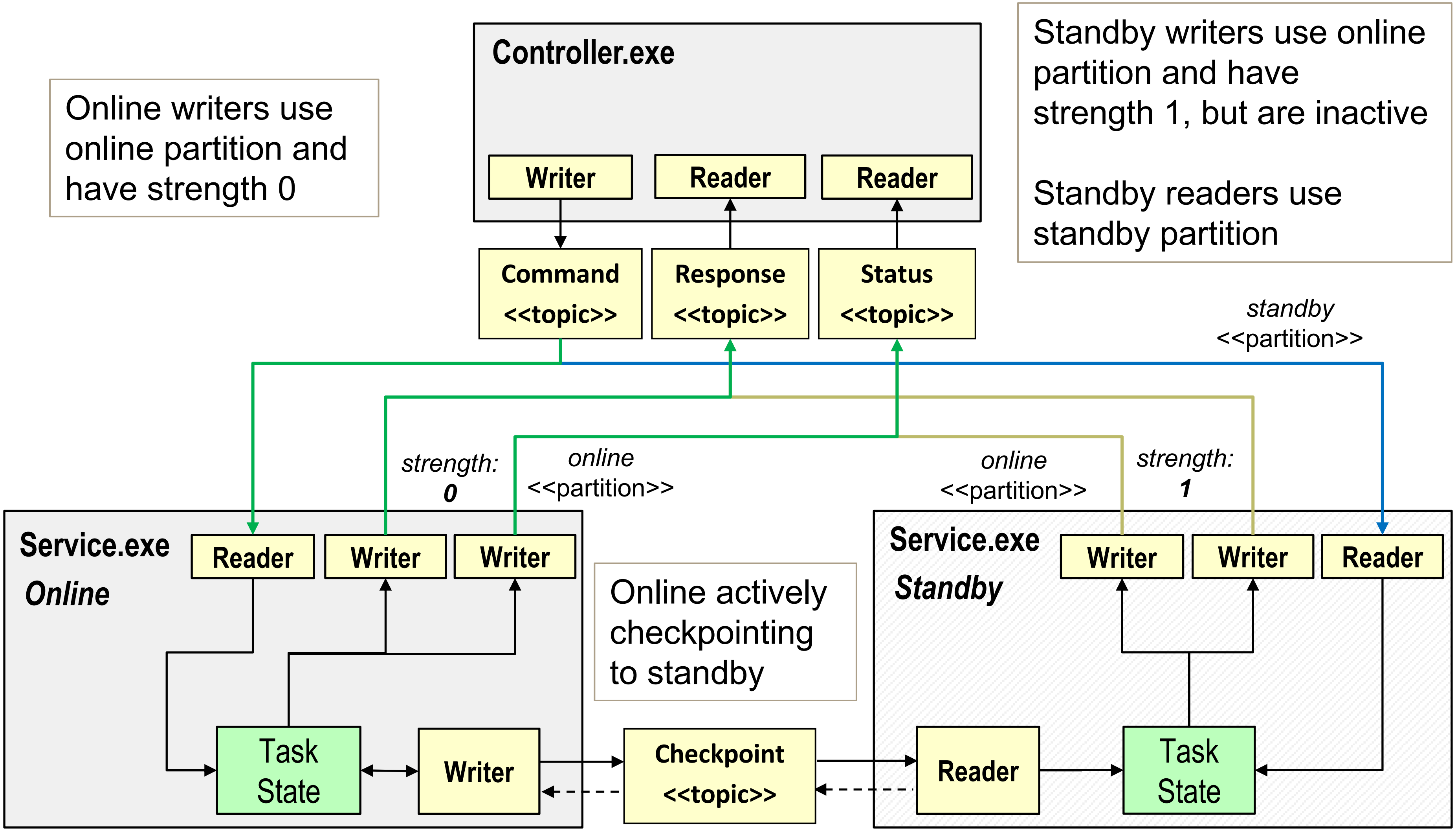
Entity Registry Handles Failover



- Entities are registered with registry during initialization
- Registry handles all entity updates as a result of state change from standby to operate
- Minimizes parts of the code which need to be modified to handle failover – separation of concerns

Fault Tolerance Example Step-by-Step

Raytheon
Integrated Defense Systems



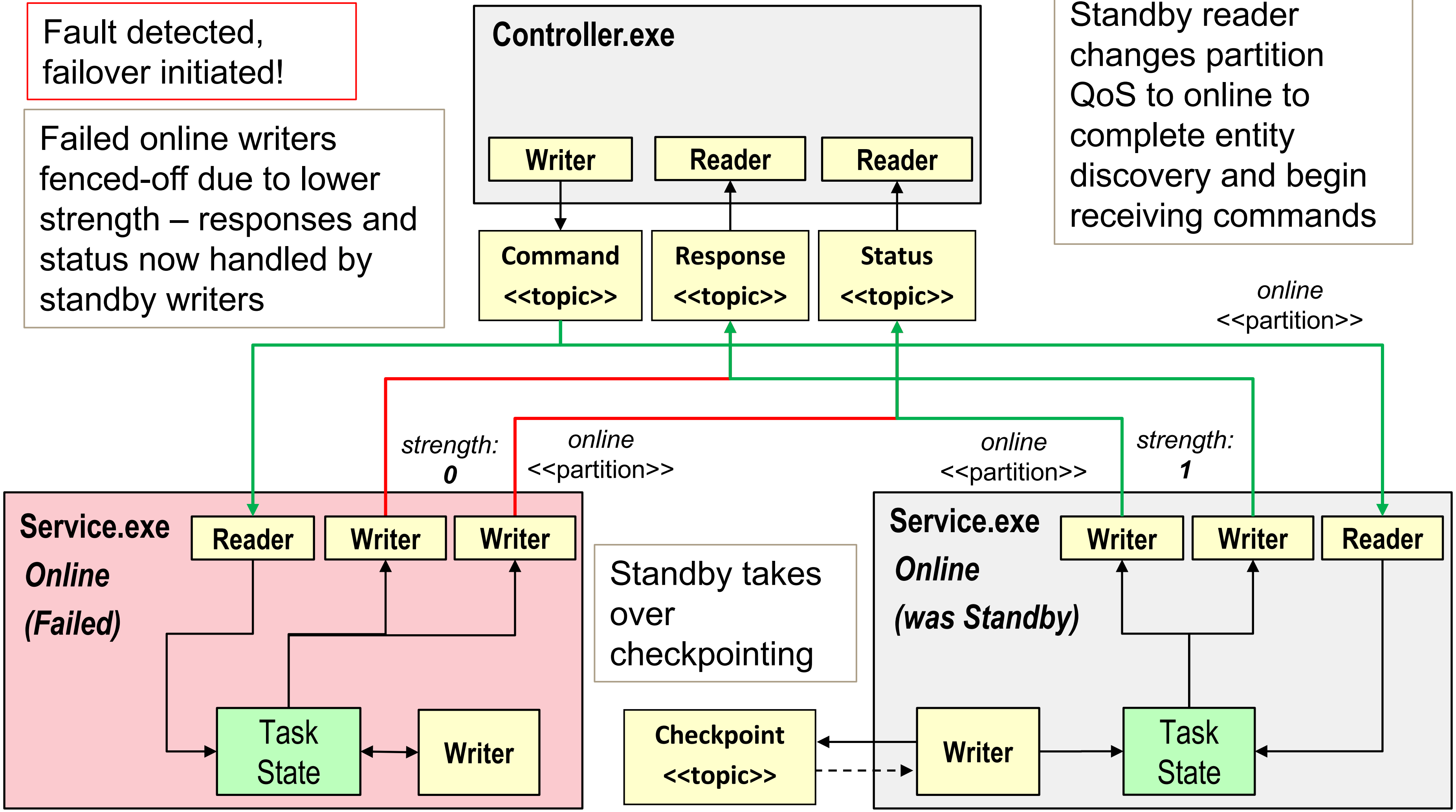
Fault Tolerance Example Step-by-Step

Raytheon
Integrated Defense Systems

Fault detected, failover initiated!

Failed online writers fenced-off due to lower strength – responses and status now handled by standby writers

Standby reader changes partition QoS to online to complete entity discovery and begin receiving commands

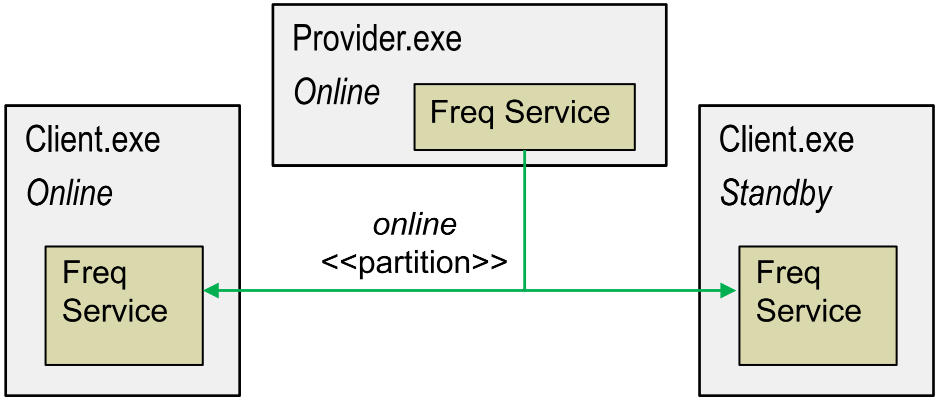


Standby takes over checkpointing

DDS Enablers of Fault Tolerance

- Decoupled publish-subscribe model
  - Built-in Discovery allows application to avoid costly connection reconfiguration during failover

- Quality of Service
  - Tunable on a per-topic, per-entity basis – so don't need a single solution for all use cases
  - Example: Standby instances of passive microservices can receive online data rather than using standby partition, simplifying fault tolerance for microservices



DDS enables effective Fault Tolerance solutions