Wait-free Hash Maps in the Entity-Component-System Pattern for Realtime Interactive Systems

Patrick Lange, Rene Weller, Gabriel Zachmann

University of Bremen, Germany

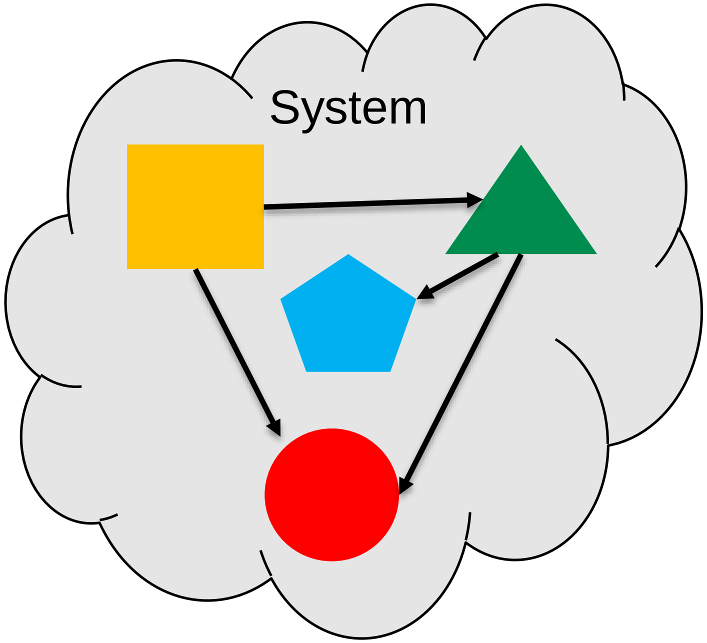
cgvr.cs.uni-bremen.de

9th SEARIS Workshop at IEEE VR

19-23 March 2016, Greenville, SC

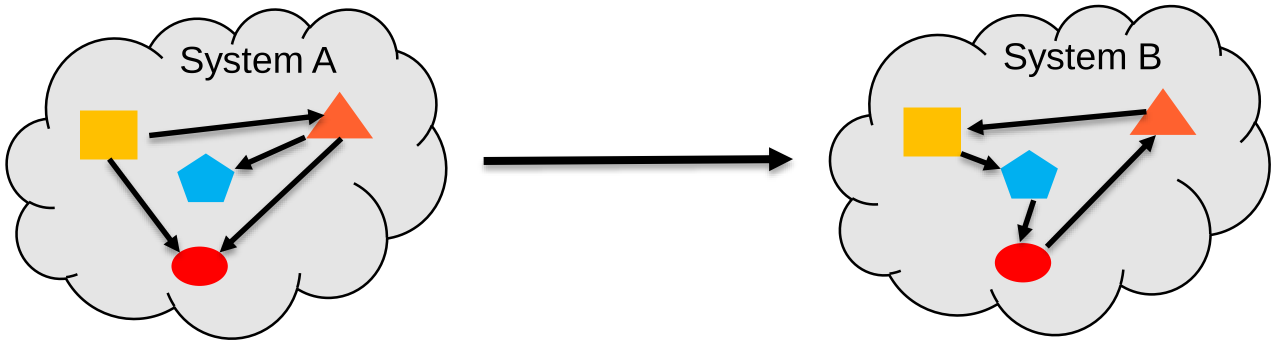
Data: Central Part in RIS Development

- Generation, management and distribution of the global simulation or world state for all software components and/or users
- Usually many independent inhomogeneous software components need to communicate and exchange data in order to generate this global state

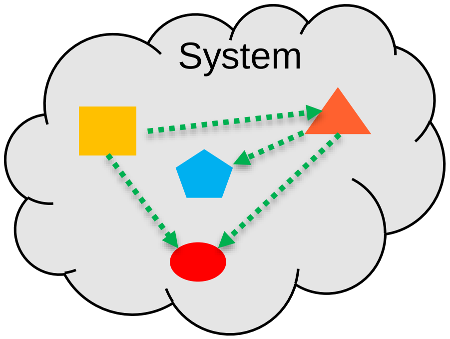


Requirements in RIS Development

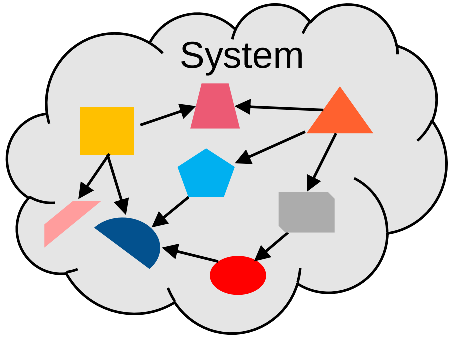
- Reusability



- (Realtime) performance

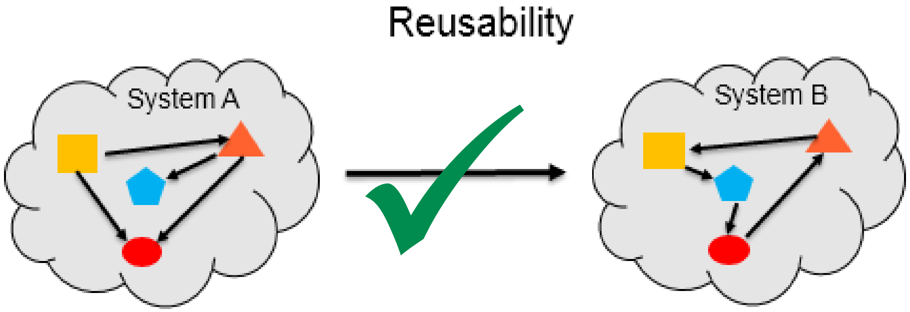


- Scalability



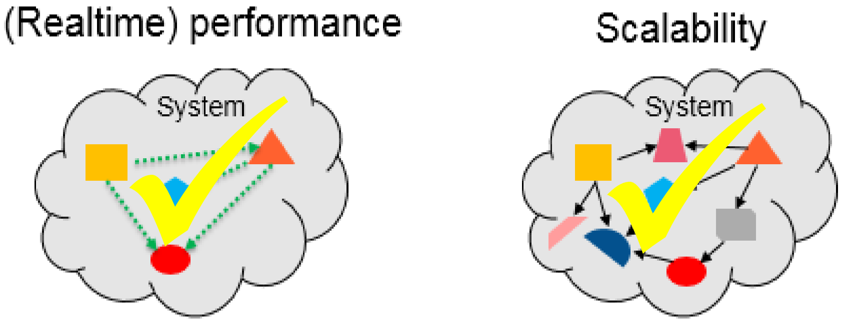
Entity-Component-System (ECS) Pattern

- Major design pattern used in modern architectures for Realtime Interactive Systems
- Strives for high reusability and architectural scalability
  - Novel architectural software concepts



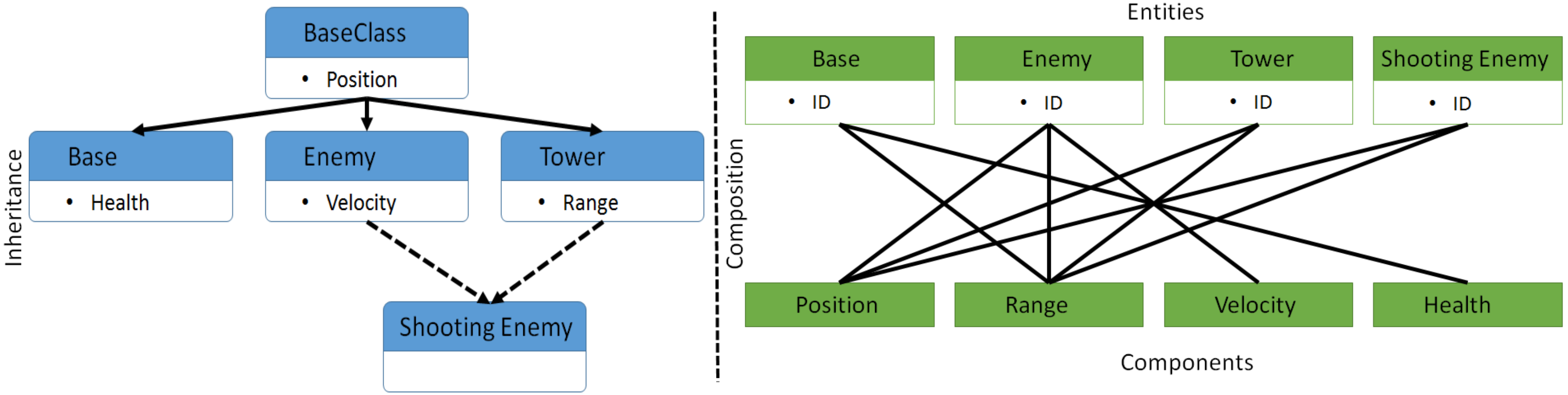
[Wiebusch'15]

- Performance and scalability for massively parallel access?



Entity-Component-System (ECS) Pattern

- Introduces three software architecture concepts
  - *Entity*: General purpose object, defined as unique id
  - *Component*: Raw data for one aspect of a general purpose object
  - *System*: Runs continuously and applies global actions on *Entities*
- Decouples high-level modules such as physics, rendering or simulation from low-level objects



ECS: Game-based Example

Systems

Physics

Input

Components

Position

Velocity

Range

Health

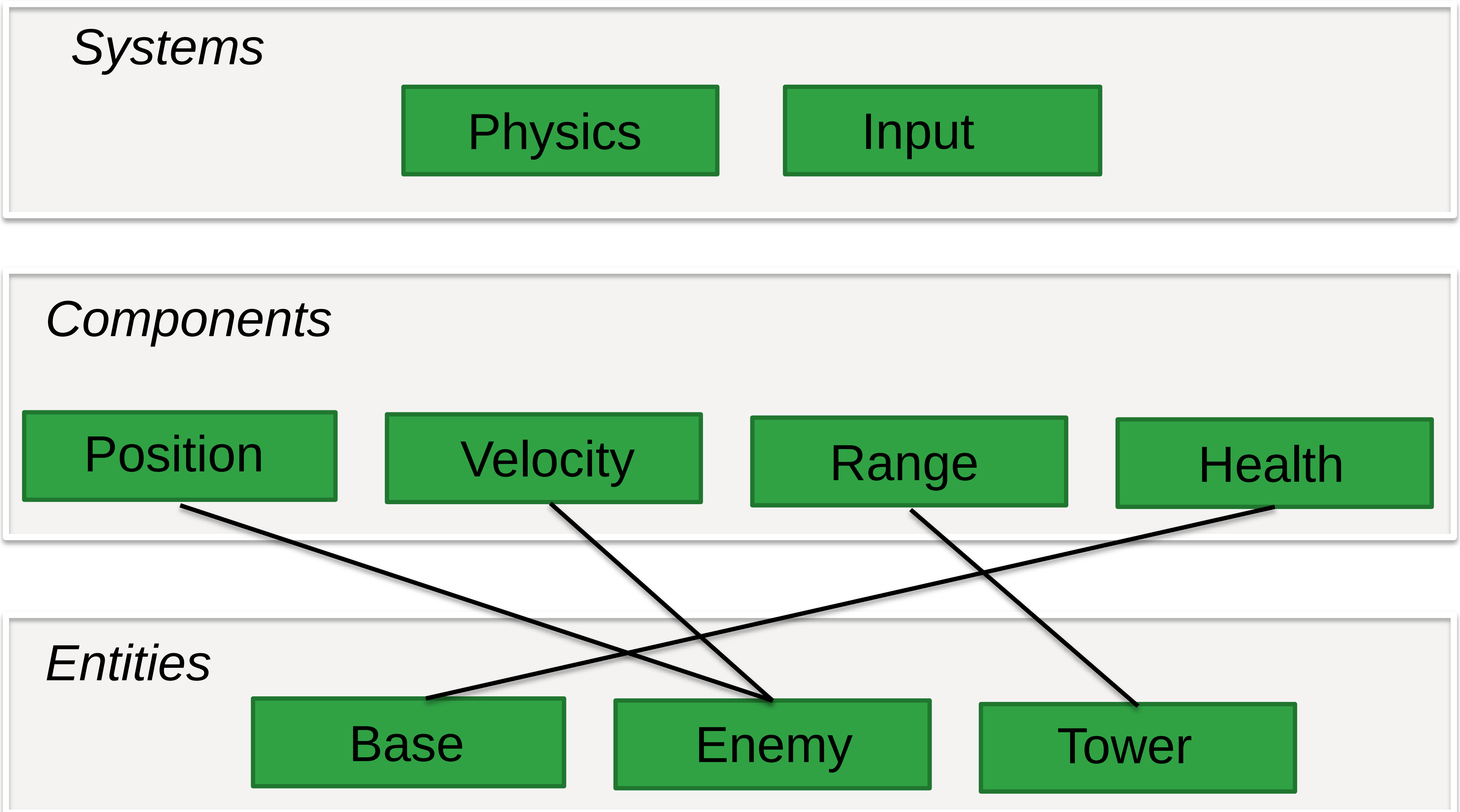
Entities

Base

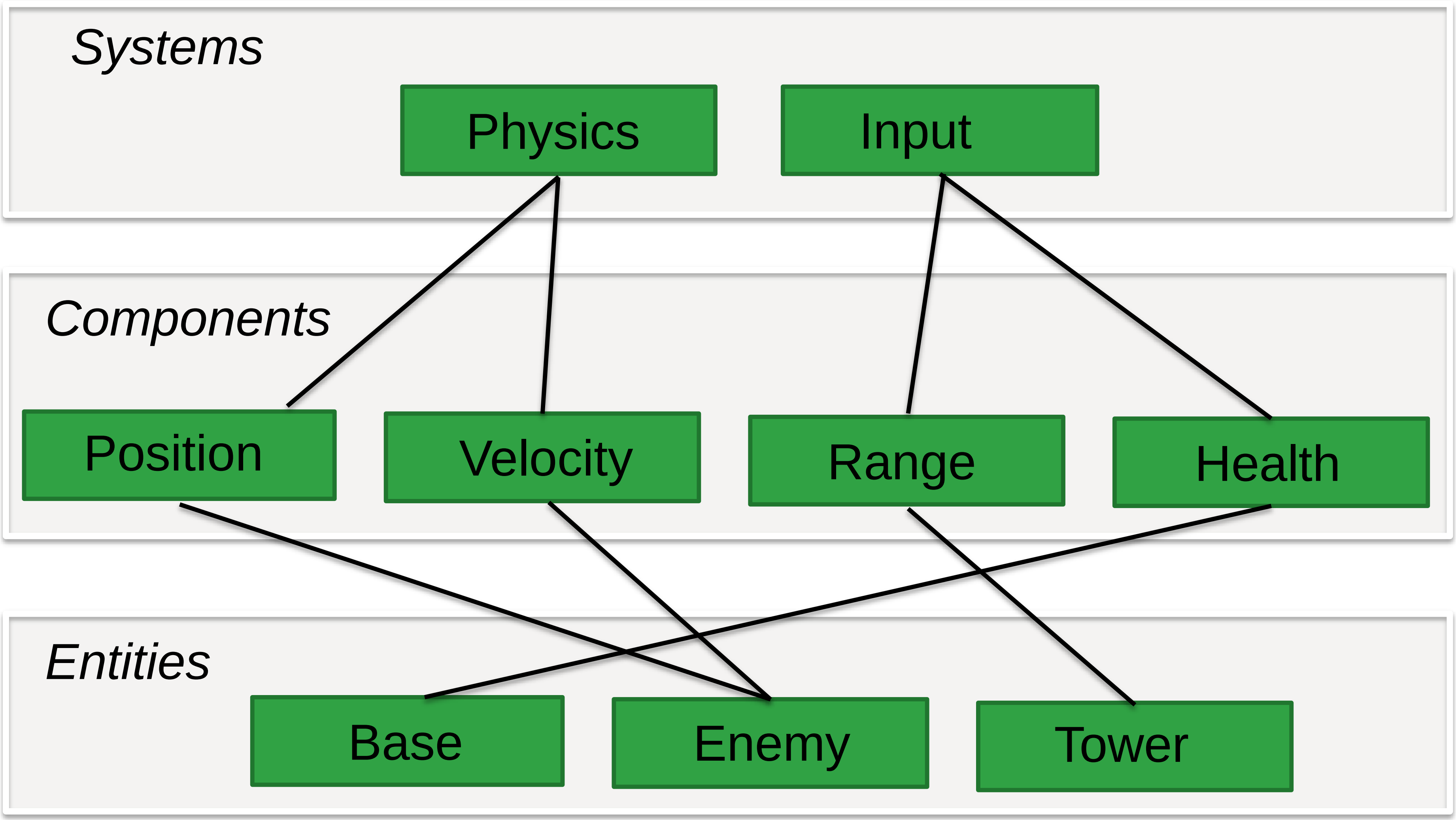
Enemy

Tower

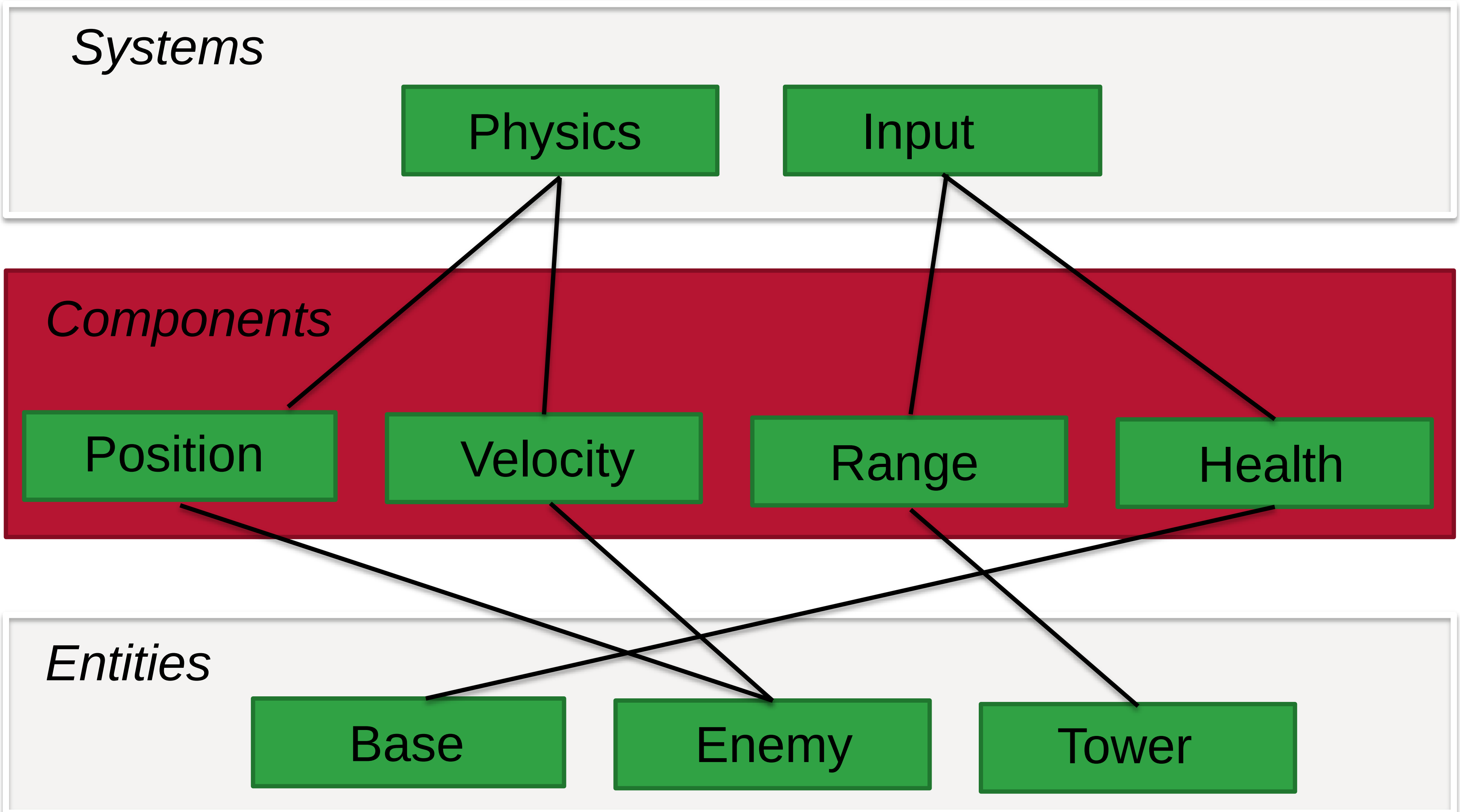
ECS: Game-based Example



ECS: Game-based Example

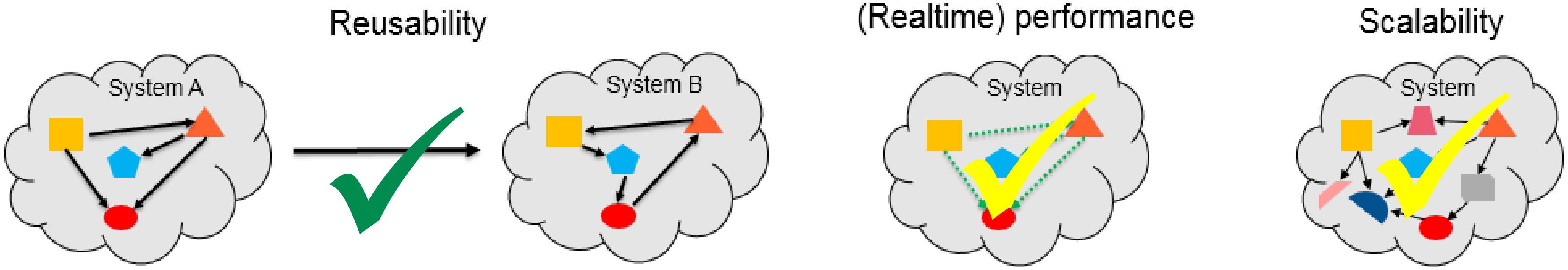


ECS: Game-based Example



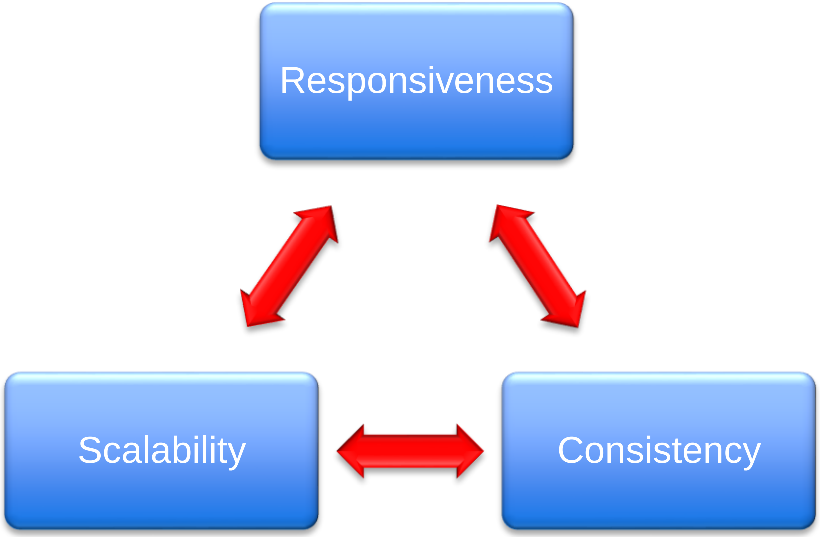
ECS: Shared Data Structures

- Current RIS applications inherit many *Entities*, *Components* and *Systems*
- Parallelization of *System* access necessary in order to preserve realtime performance constraints
  - The container of *Components* becomes a shared data structure
- ECS does not give guidelines or specification how to solve this problem



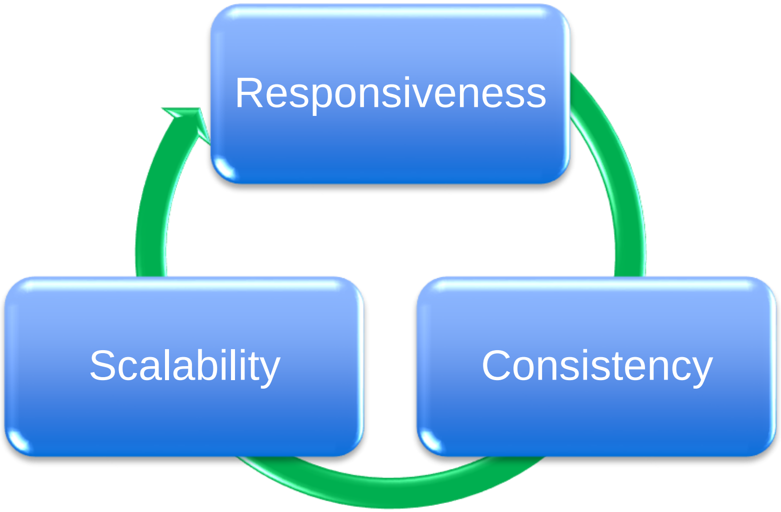
Concurrency Control for RIS

- Process of managing simultaneous execution of software components on shared global word/simulation state
- RIS reserach concerns low-level concepts and high-level concepts for parallelism [Latoschik'11,Rehfeld'13,Knot'14]
- High-performance architectures for e.g. sophisticated (3D) simulations (C/C++, CUDA, OpenMP, OpenGL..)



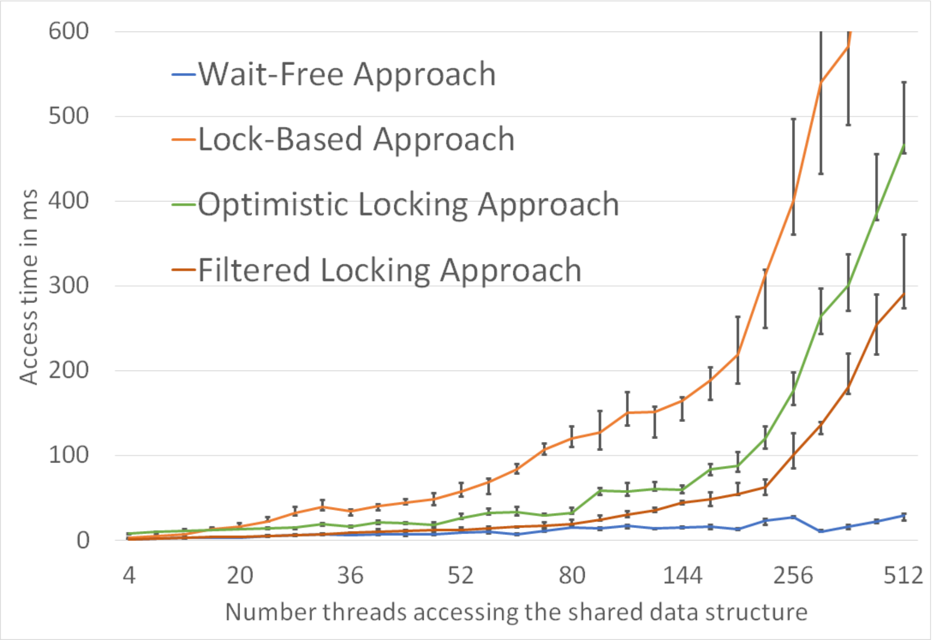
Wait-free Hash Maps

- Guarantee access to a shared data structure in a finite number of steps (e.g. as traditional thread or OpenMP implementation)
- Does not need any traditional locking mechanism
- Deliver high performance even for massive concurrent access



Wait-free Hash Maps: Basic Idea

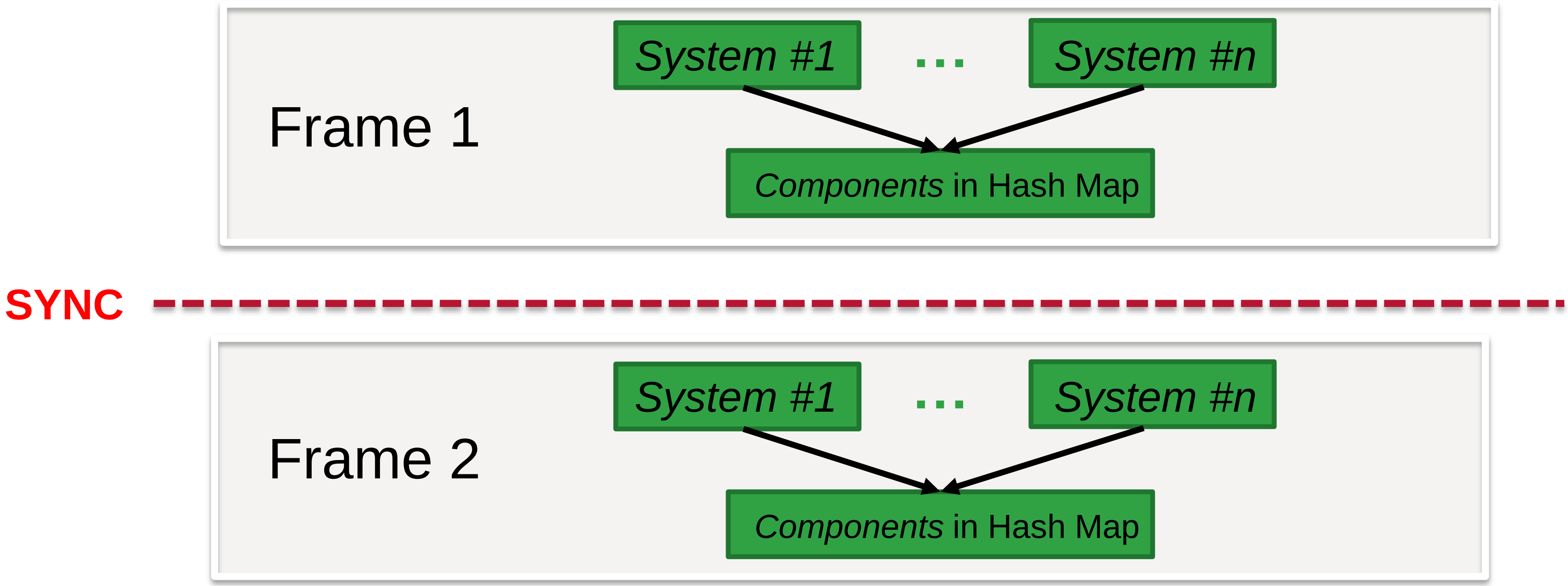
- Assignment of unique identifiers to each data packet which is exchanged between software components
- Every data packet is stored inside a hash map which resembles the complete system state
- De-coupling and parallelization of read, write and data deletion processes via atomic operations and memory cloning [Lange‘14, Lange‘15]



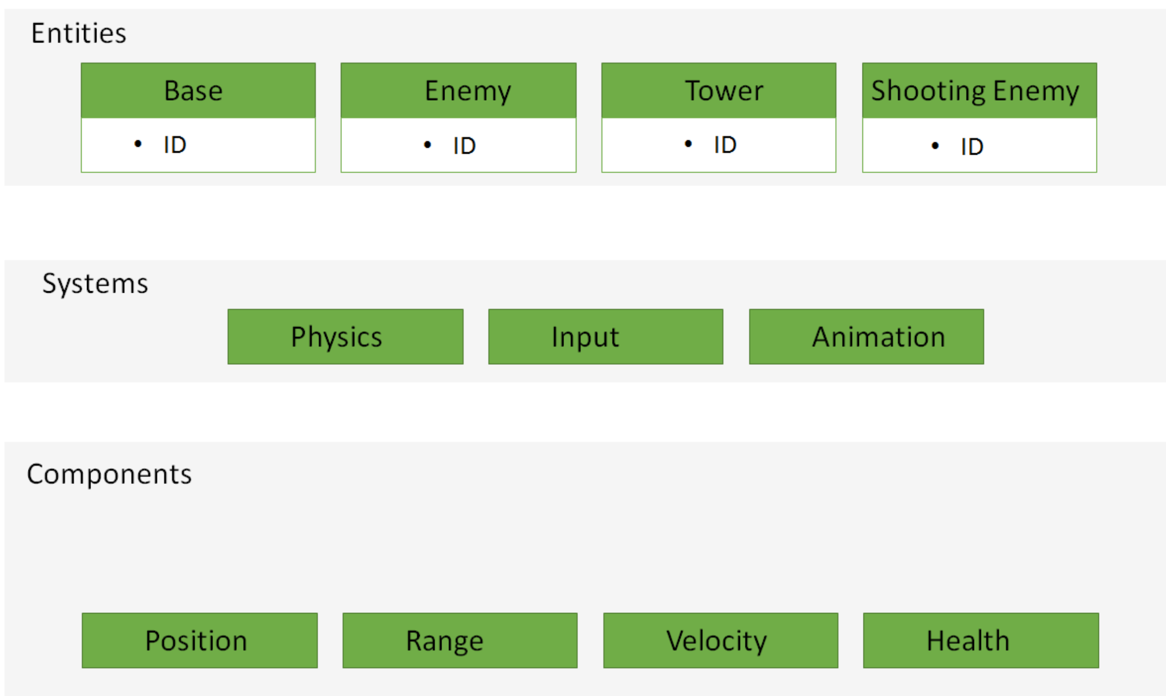
[Adapted from Lange’15]

Wait-free Hash Maps: Applications

- Massive concurrent access (> 50 threads) per simulation/system frame
  - Multi-agent system based simulation, simulation-based optimization

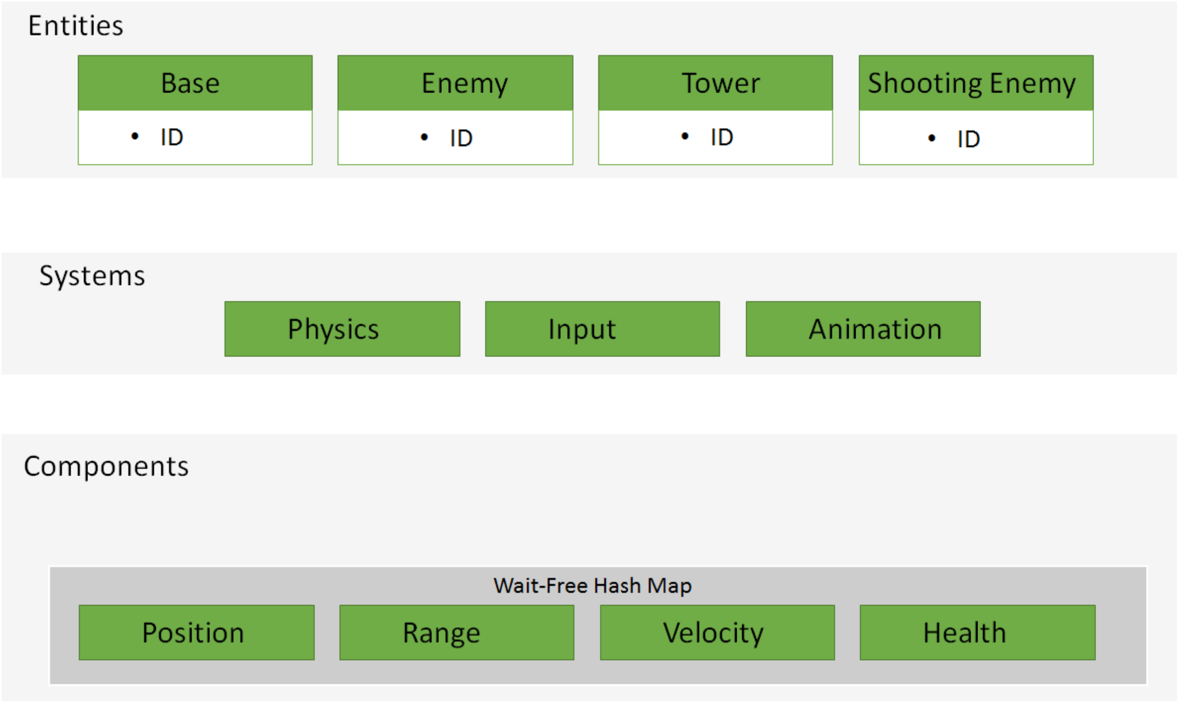


Integration of Wait-free Hash Maps



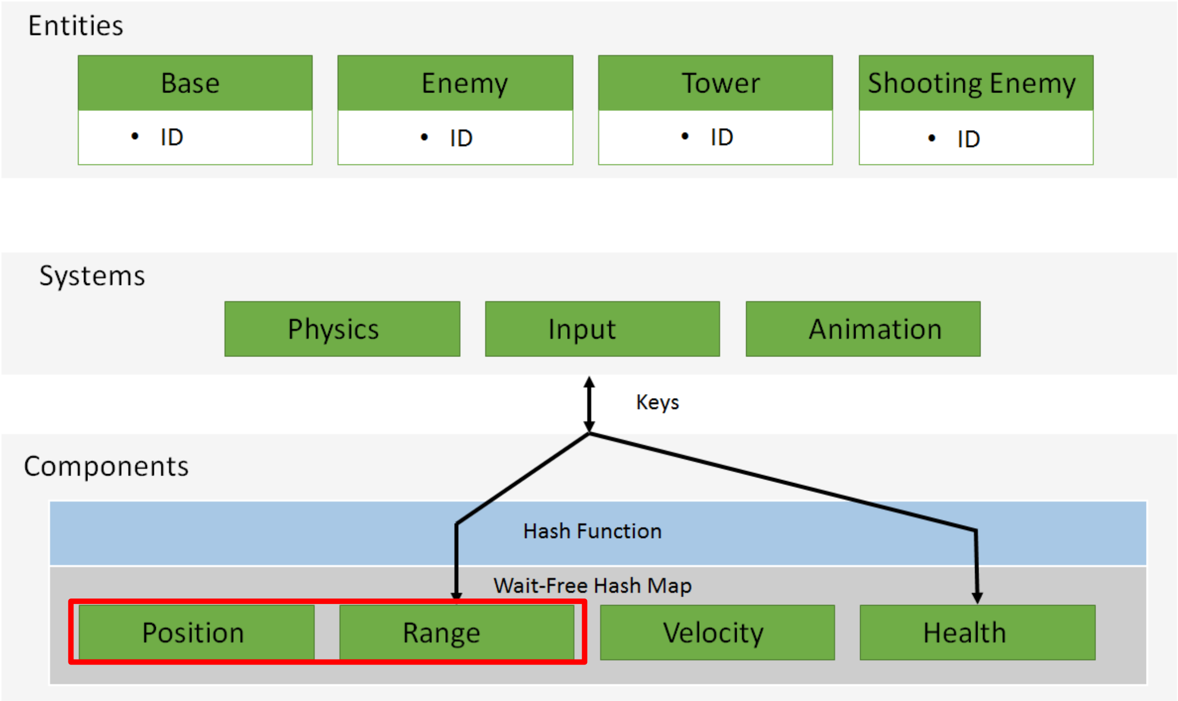
Integration of Wait-free Hash Maps

- All *Components* reside in our wait-free hash map



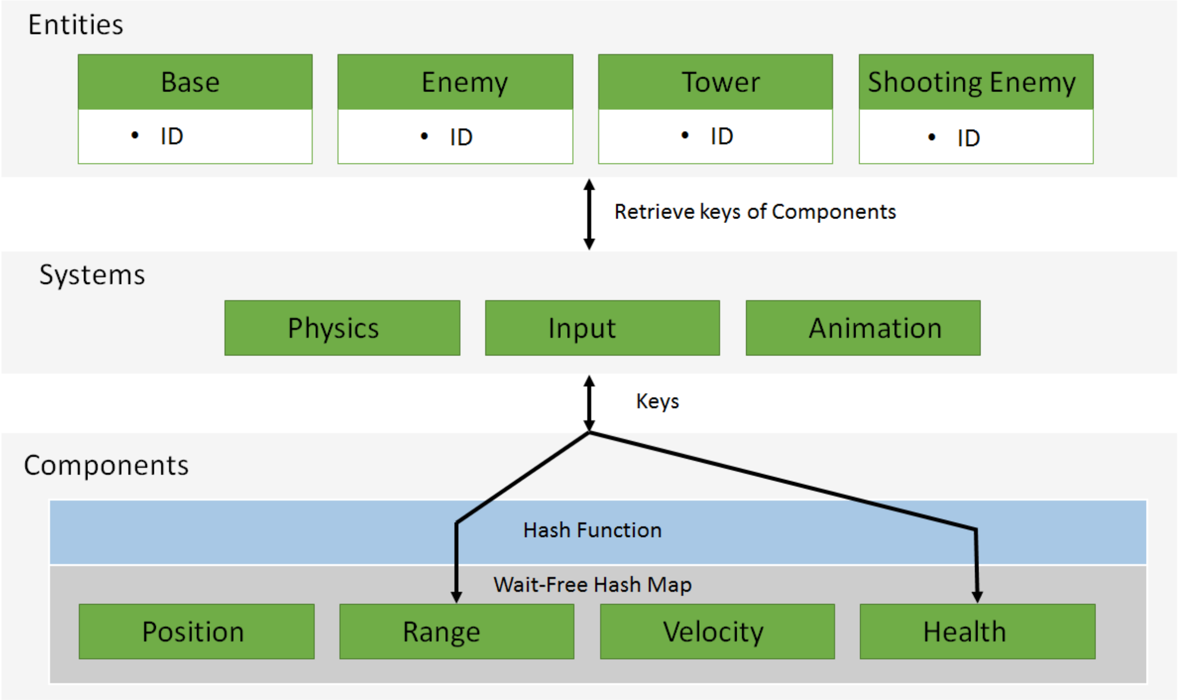
Integration of Wait-free Hash Maps

- All *Components* reside in our wait-free hash map
- *Components* (also collections) are accessible via unique keys



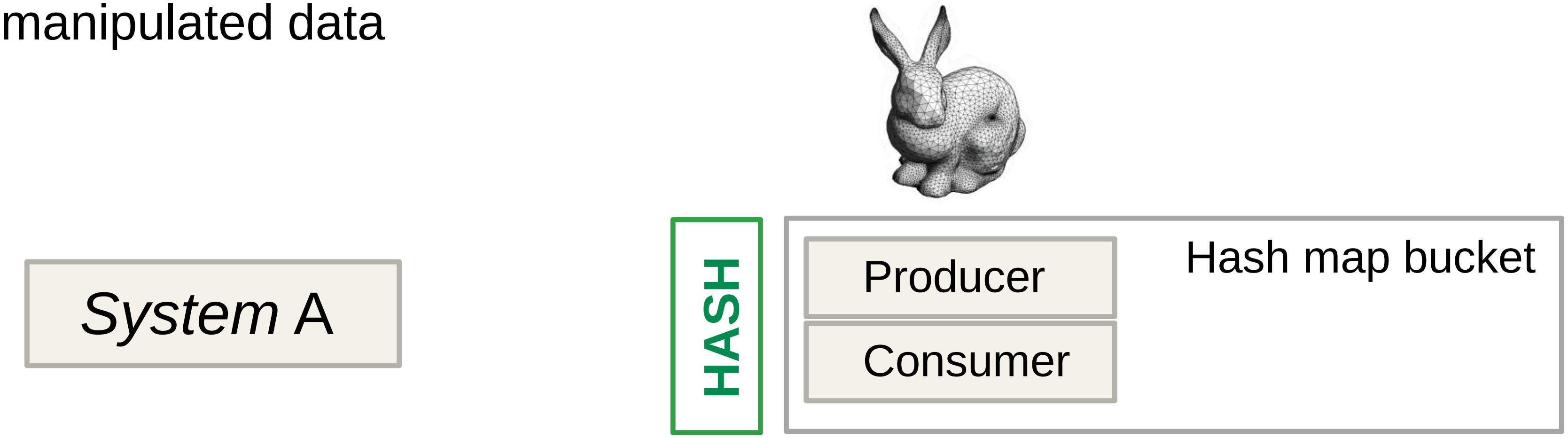
Integration of Wait-free Hash Maps

- All *Components* reside in our wait-free hash map
- *Components* are accessible via unique keys
- *Entity* composition as list of *Component* keys



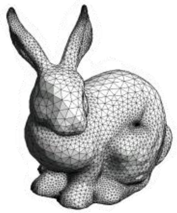
Wait-free Hash Maps: Double Buffering

- Producer and consumer version of data within hash map
  - Atomic reference counter guards consumer versions
- Every write access to the hash map generates a clone of the manipulated data



Wait-free Hash Maps: Double Buffering

- Producer and consumer version of data within hash map
  - Atomic reference counter guards consumer versions
- Every write access to the hash map generates a clone of the manipulated data



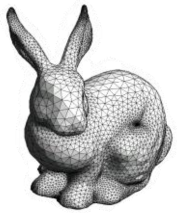
System A —GET(KEY)→ Consumer

Producer

Consumer

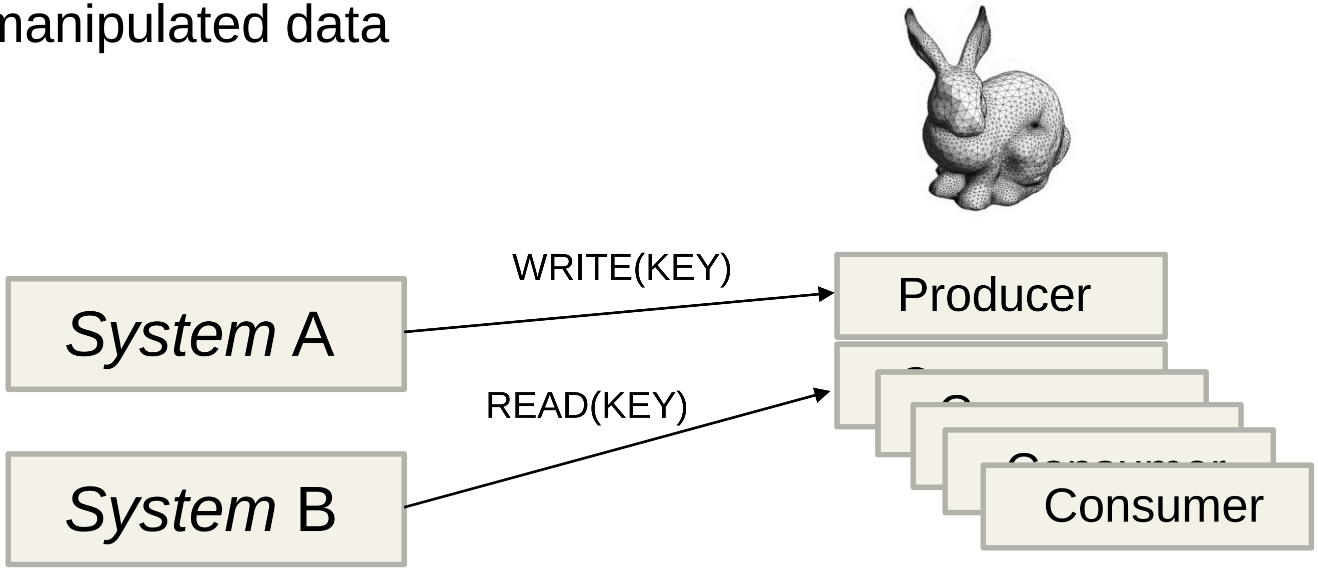
Wait-free Hash Maps: Double Buffering

- Producer and consumer version of data within hash map
  - Atomic reference counter guards consumer versions
- Every write access to the hash map generates a clone of the manipulated data



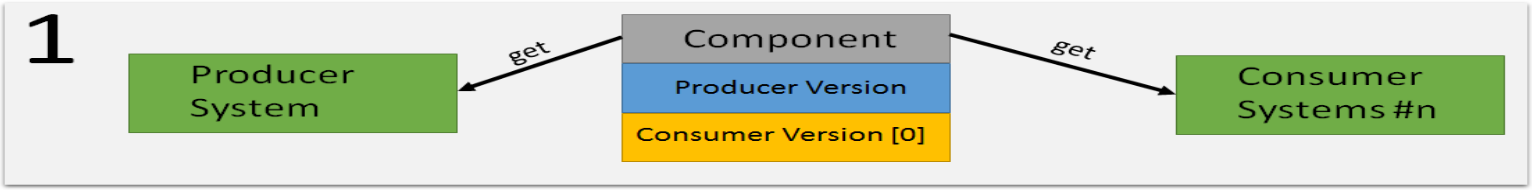
Wait-free Hash Maps: Double Buffering

- Producer and consumer version of data within hash map
  - Atomic reference counter guards consumer versions
- Every write access to the hash map generates a clone of the manipulated data

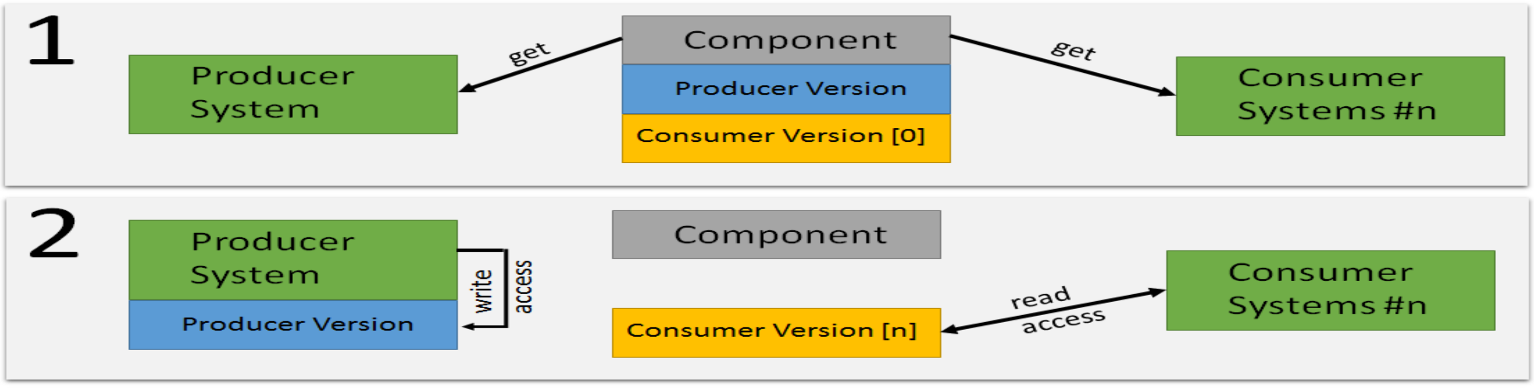


- Parallel read access can return, in accordance to RIS setup, any old state

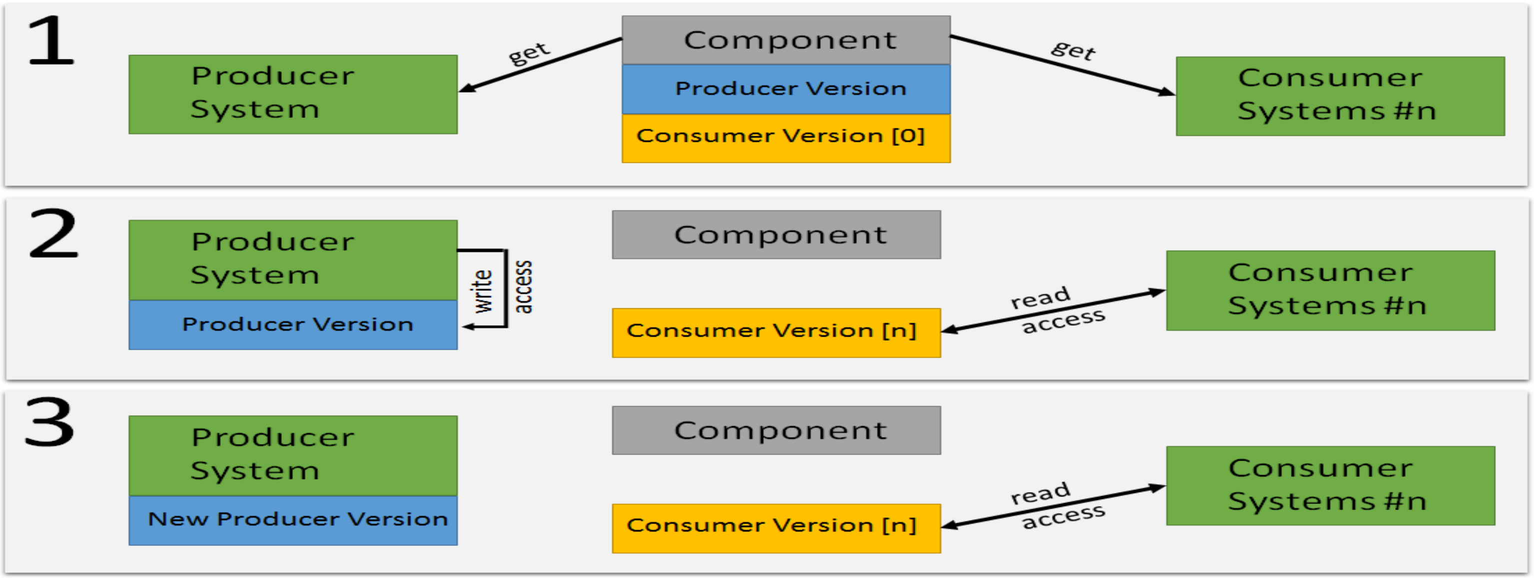
Integration of Wait-free Hash Maps



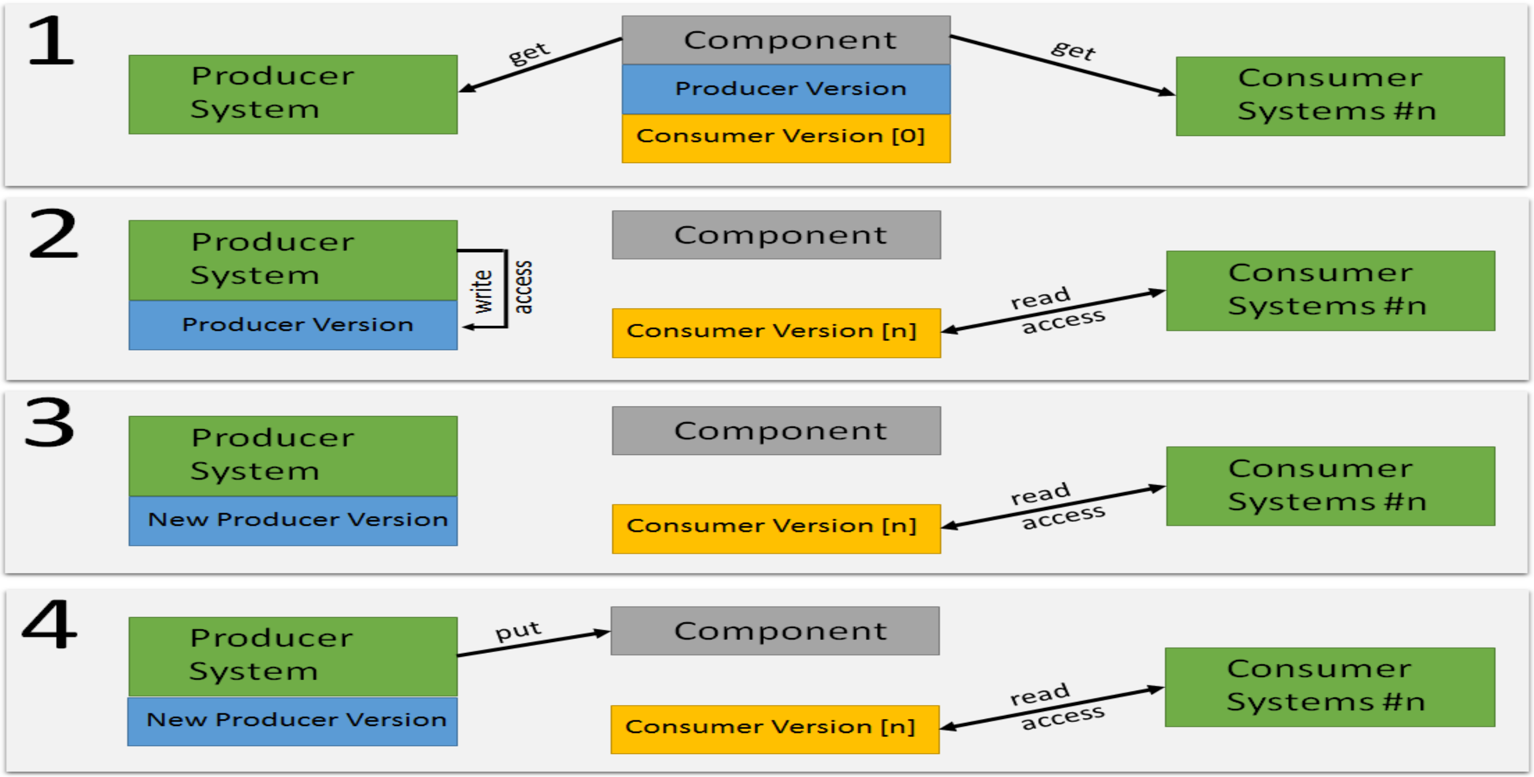
Integration of Wait-free Hash Maps



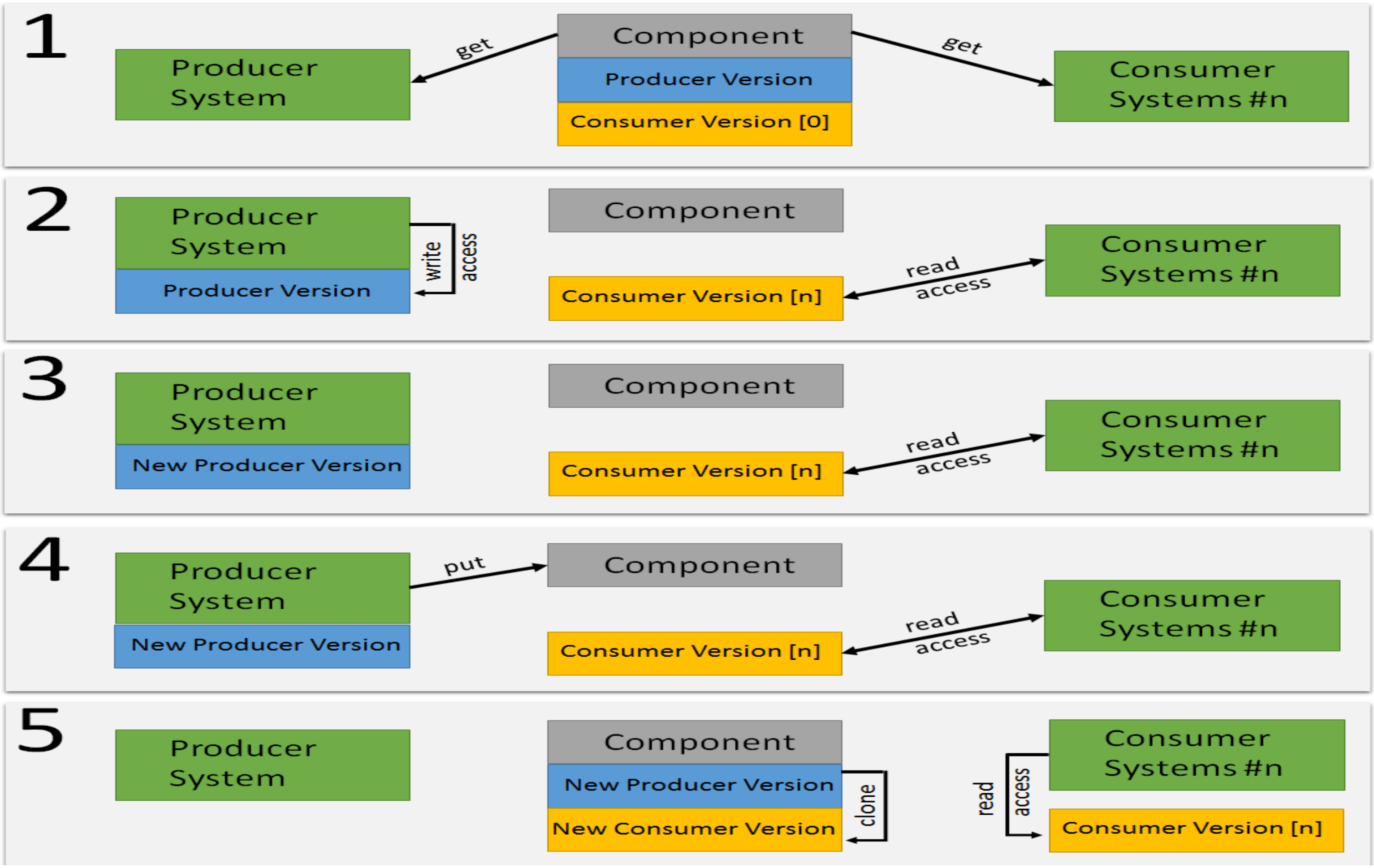
Integration of Wait-free Hash Maps



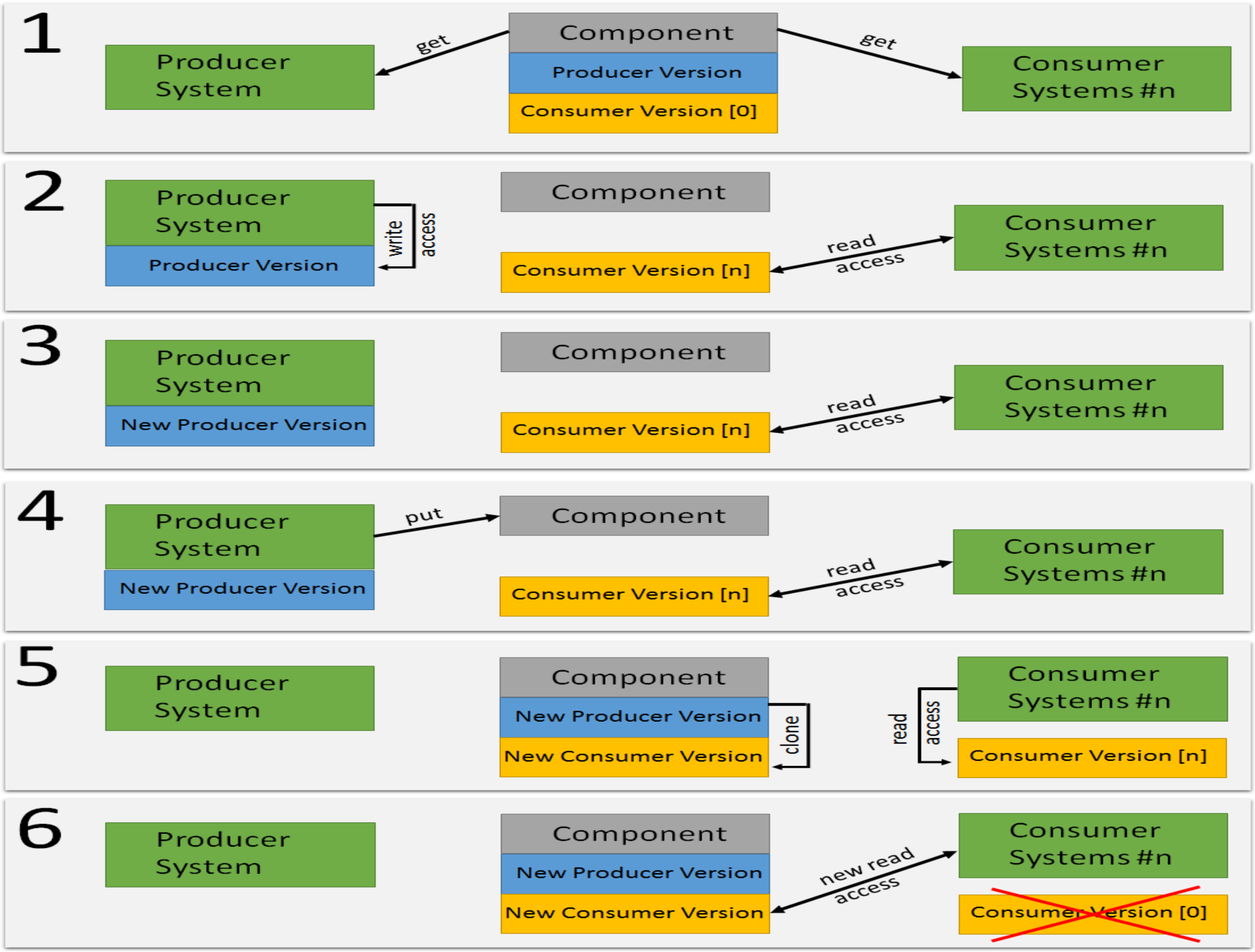
Integration of Wait-free Hash Maps



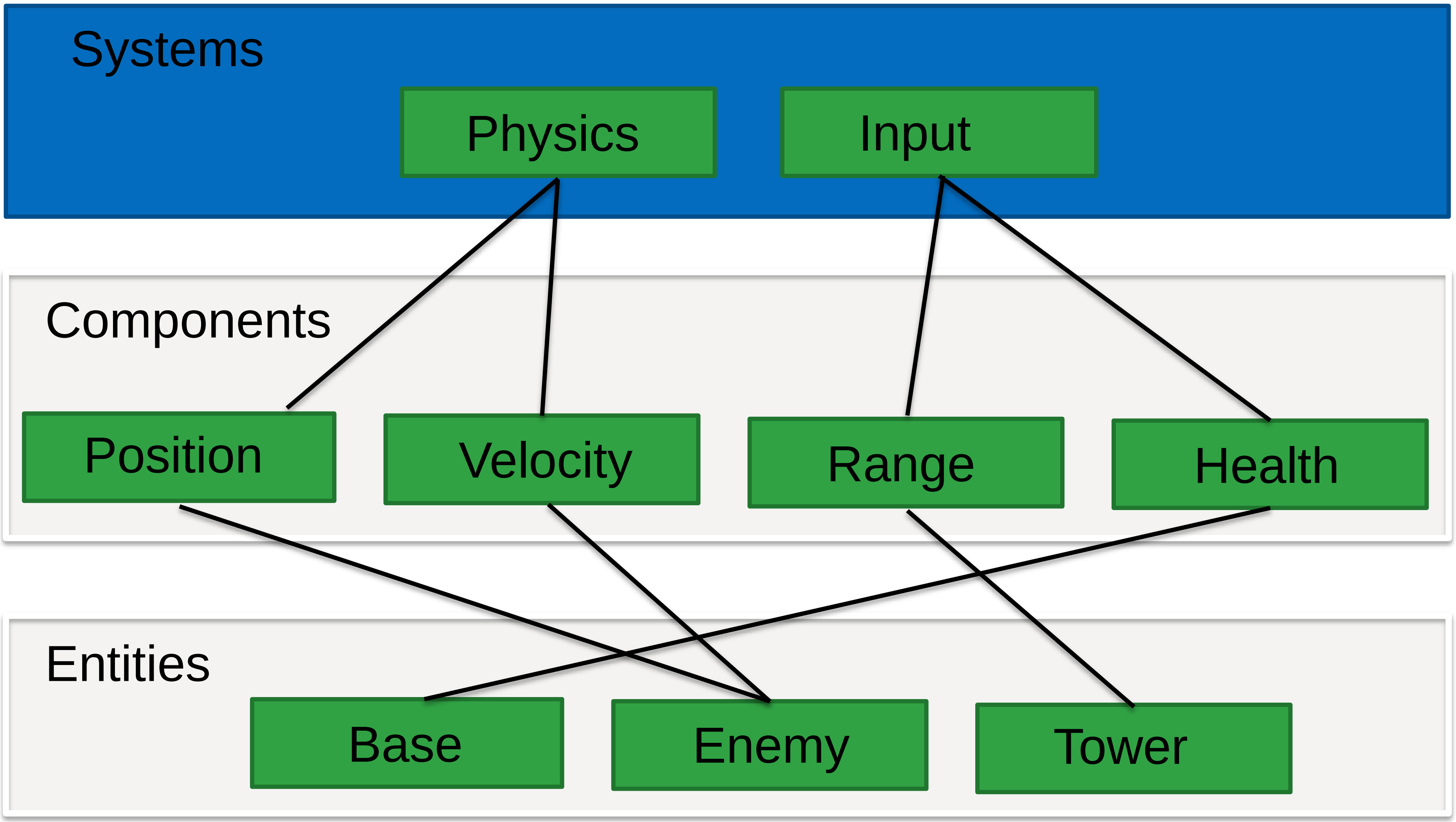
Integration of Wait-free Hash Maps



Integration of Wait-free Hash Maps



Integration of Wait-free Hash Maps



Integration of Wait-free Hash Maps

```
//Define OpenMP parallelization with x threads
#pragma omp parallel for num_threads(x)
for( all Entities of System )
{



}
```

Integration of Wait-free Hash Maps

```
//Define OpenMP parallelization with x threads
#pragma omp parallel for num_threads(x)
for( all Entities of System )
{
    for( all WriteKeys of Entity )
    {
        Component = Hashmap.get(WriteKey)
        // Change component
        // ….
        Clone = Hashmap.put(Component, WriteKey)
    }
}
```

Bremen

Integration of Wait-free Hash Maps

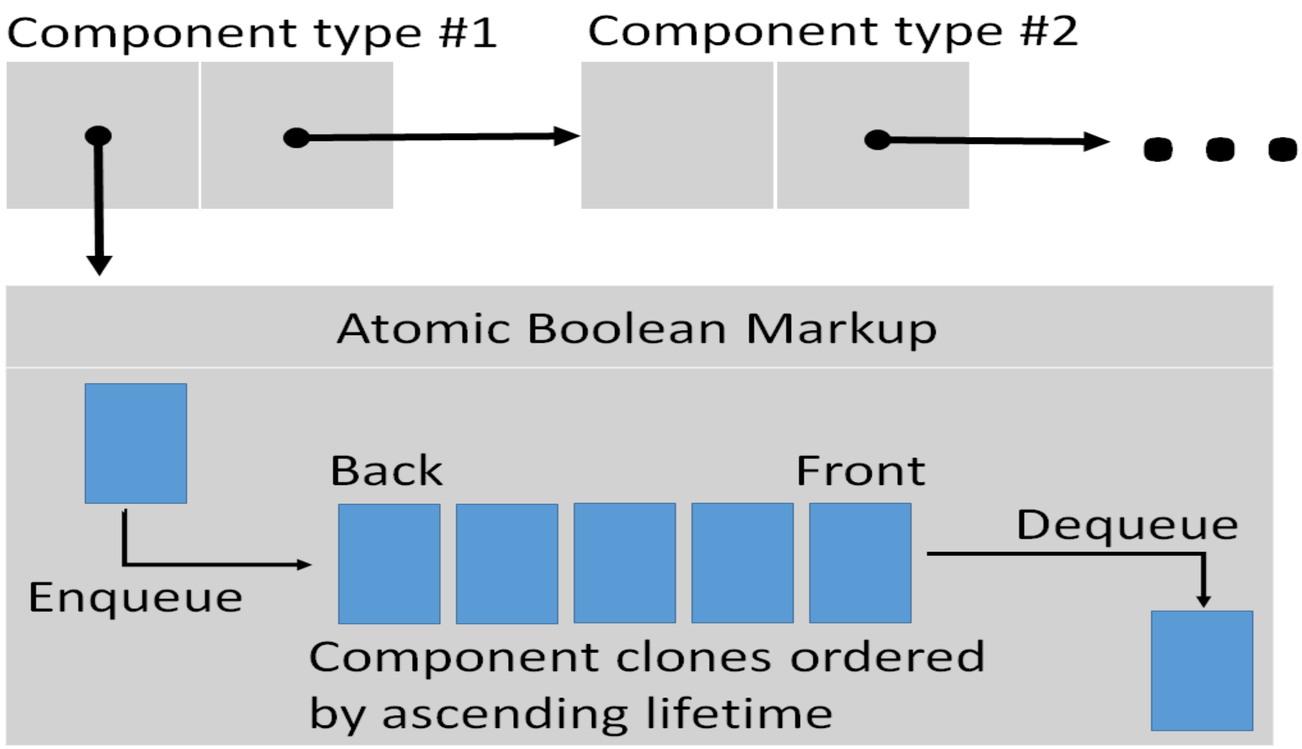
```
//Define OpenMP parallelization with x threads
#pragma omp parallel for num_threads(x)
for( all Entities of System )
{
    for( all WriteKeys of Entity )
    {
        Component = Hashmap.get(WriteKey)
        // Change component
        // ….
        Clone = Hashmap.put(Component, WriteKey)
    }
    for( all ReadKeys of Entity )
    {
        Component = Hashmap.get(ReadKey)
        …..
    }
}
```

Integration of Wait-free Hash Maps

```
//Define OpenMP parallelization with x threads
#pragma omp parallel for num_threads(x)
for( all Entities of System )
{
    for( all WriteKeys of Entity )
    {
        Component = Hashmap.get(WriteKey)
        // Change component
        // ….
        Clone = Hashmap.put(Component, WriteKey)
    }
    for( all ReadKeys of Entity )
    {
        Component = Hashmap.get(ReadKey)
        …..
    }
}
```

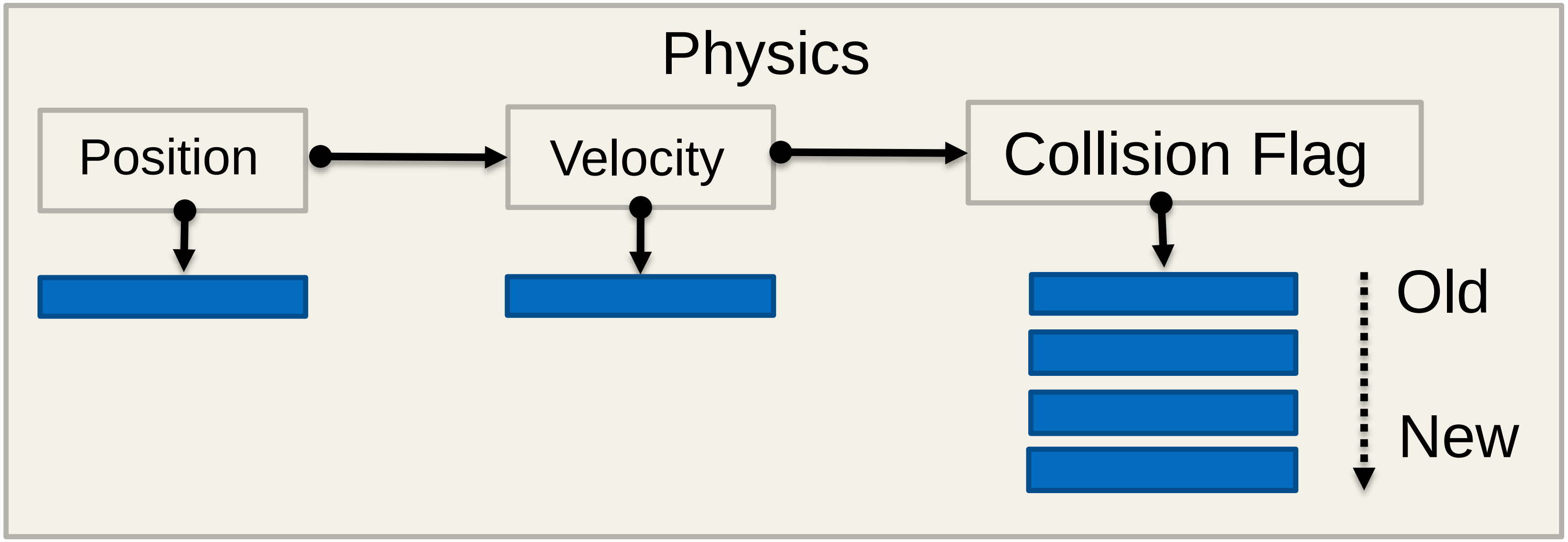
Component-wise Queues

- Different *Components* are more frequently used than other *Components*
  - *Collision detection (1000 Hz) vs. animation (30 Hz)*



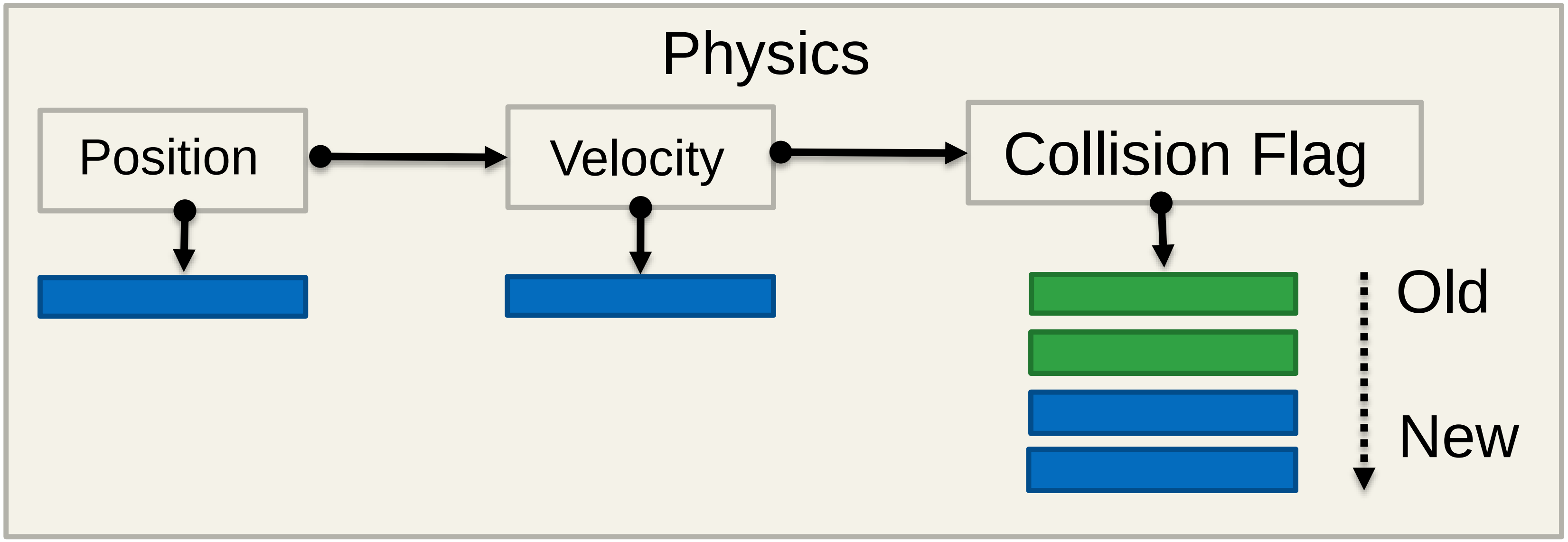
Component-wise Queues: Example

- At startup: Create *Component*-type sorted list
- Sort created cloned *Components* into corresponding queues for each *Component*-type
- Each list node contains markup for changes within queue
- Iteration checks every node for markup and queues



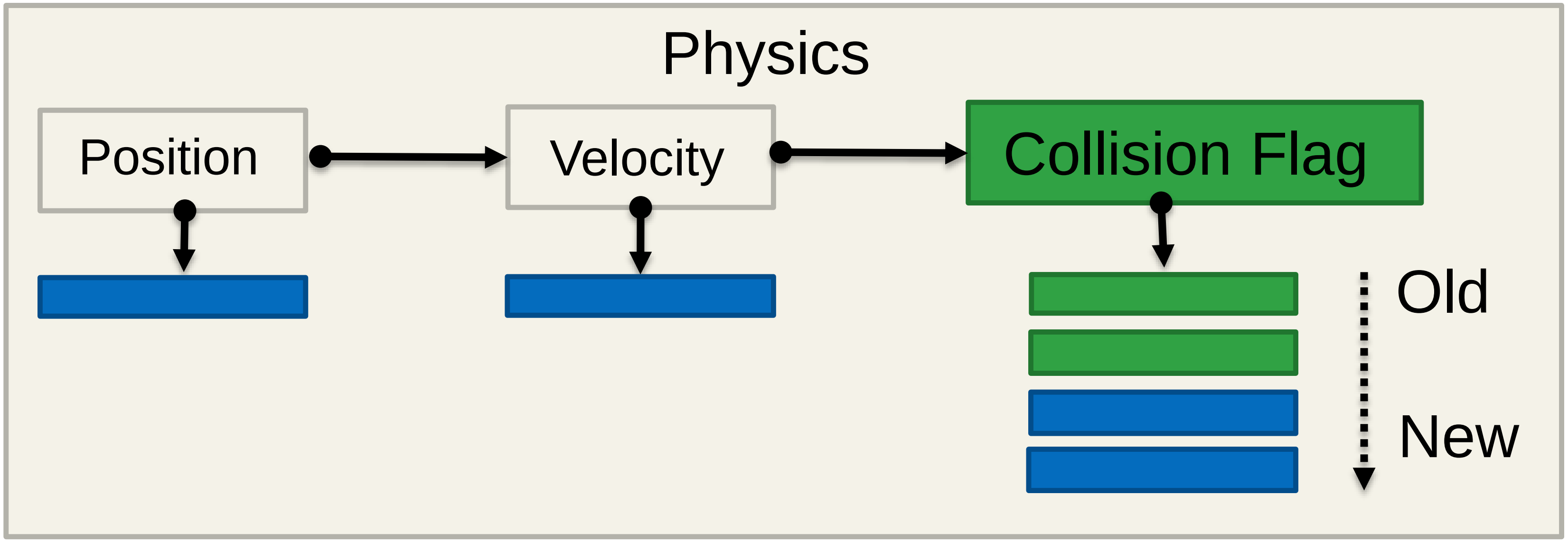
Component-wise Queues: Example

- At startup: Create *Component*-type sorted list
- Sort created cloned *Components* into corresponding queues for each *Component*-type
- Each list node contains markup for changes within queue
- Iteration checks every node for markup and queues



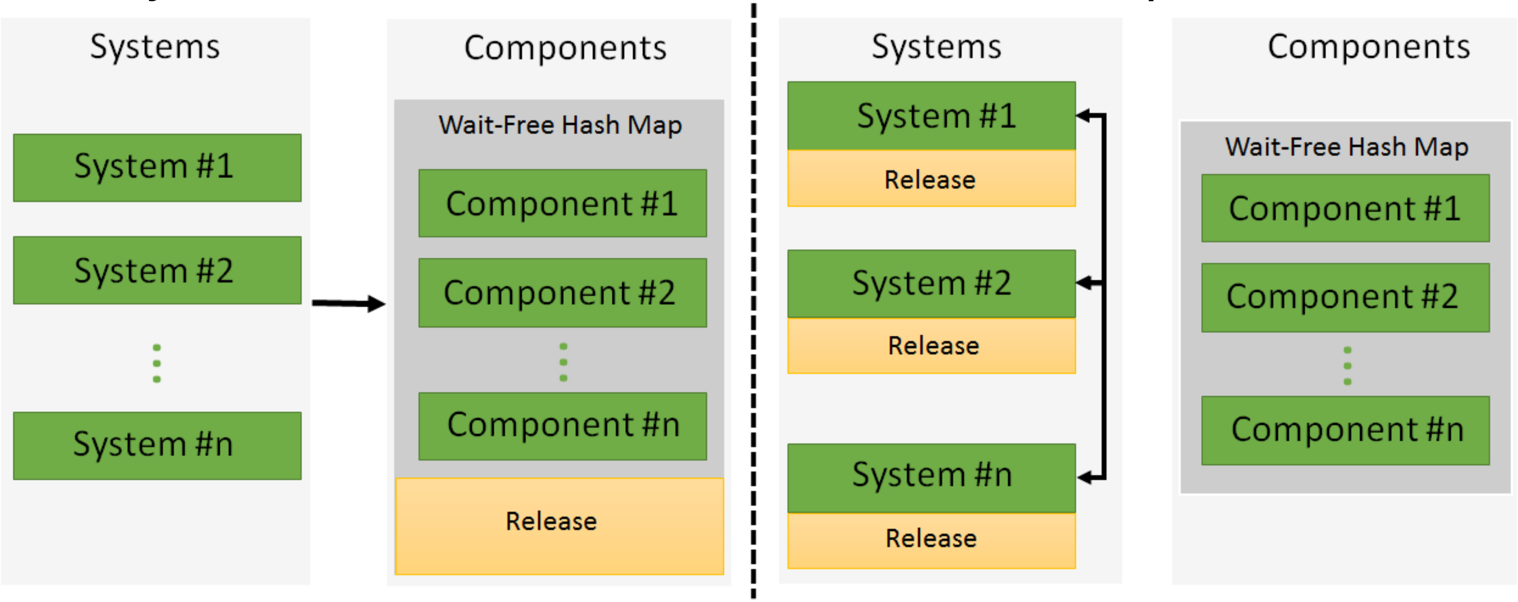
Component-wise Queues: Example

- At startup: Create *Component*-type sorted list
- Sort created cloned *Components* into corresponding queues for each *Component*-type
- Each list node contains markup for changes within queue
- Iteration checks every node for markup and queues



Memory Management

- *Component*-wise queues are either located inside hash map (centralized) or *System* implementation (decentralized)
  - Centralized in three variations: Frequency-based, continously threaded, threaded on-demand
- Rely on read access notifications via atomic operations

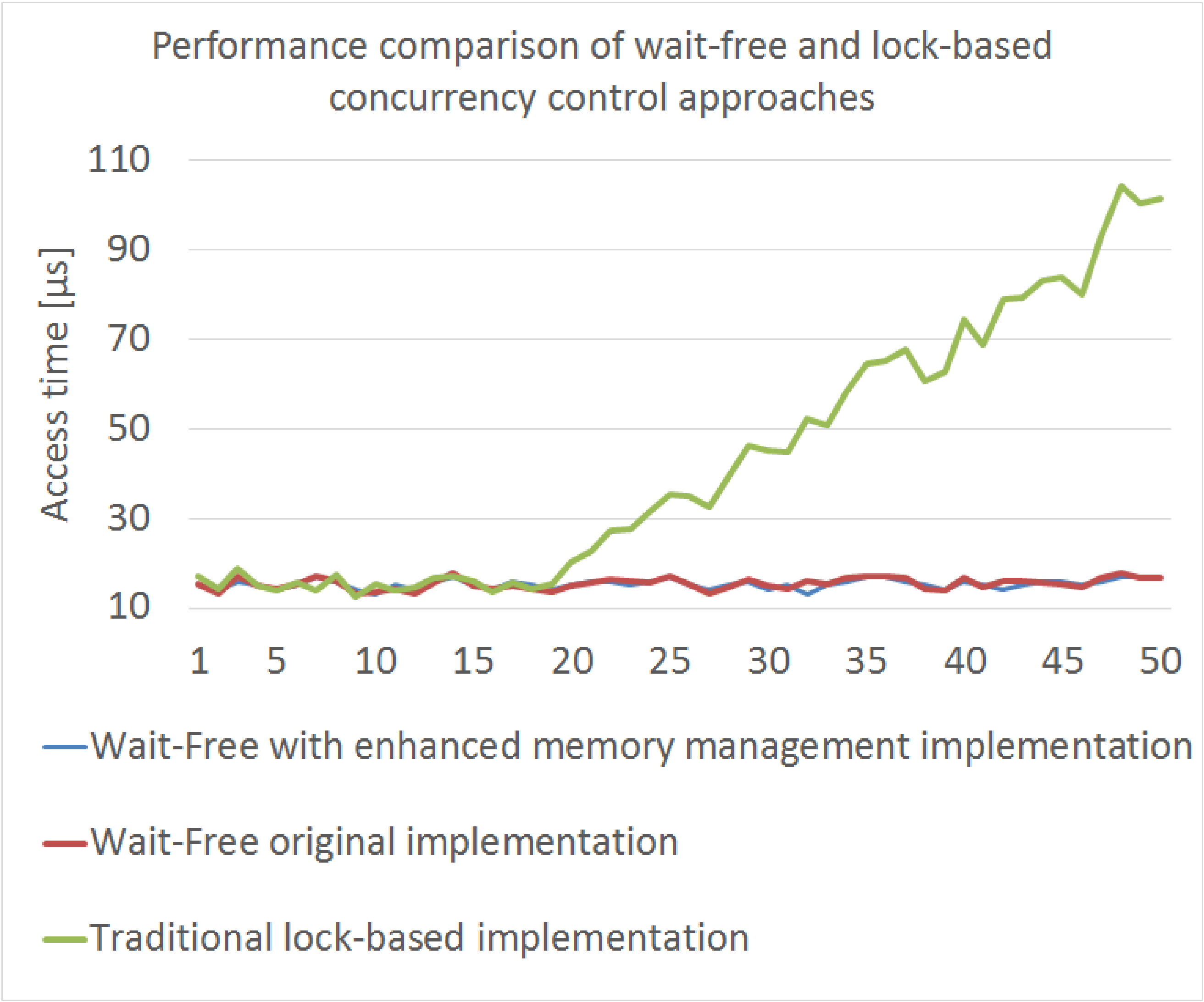


Evaluation

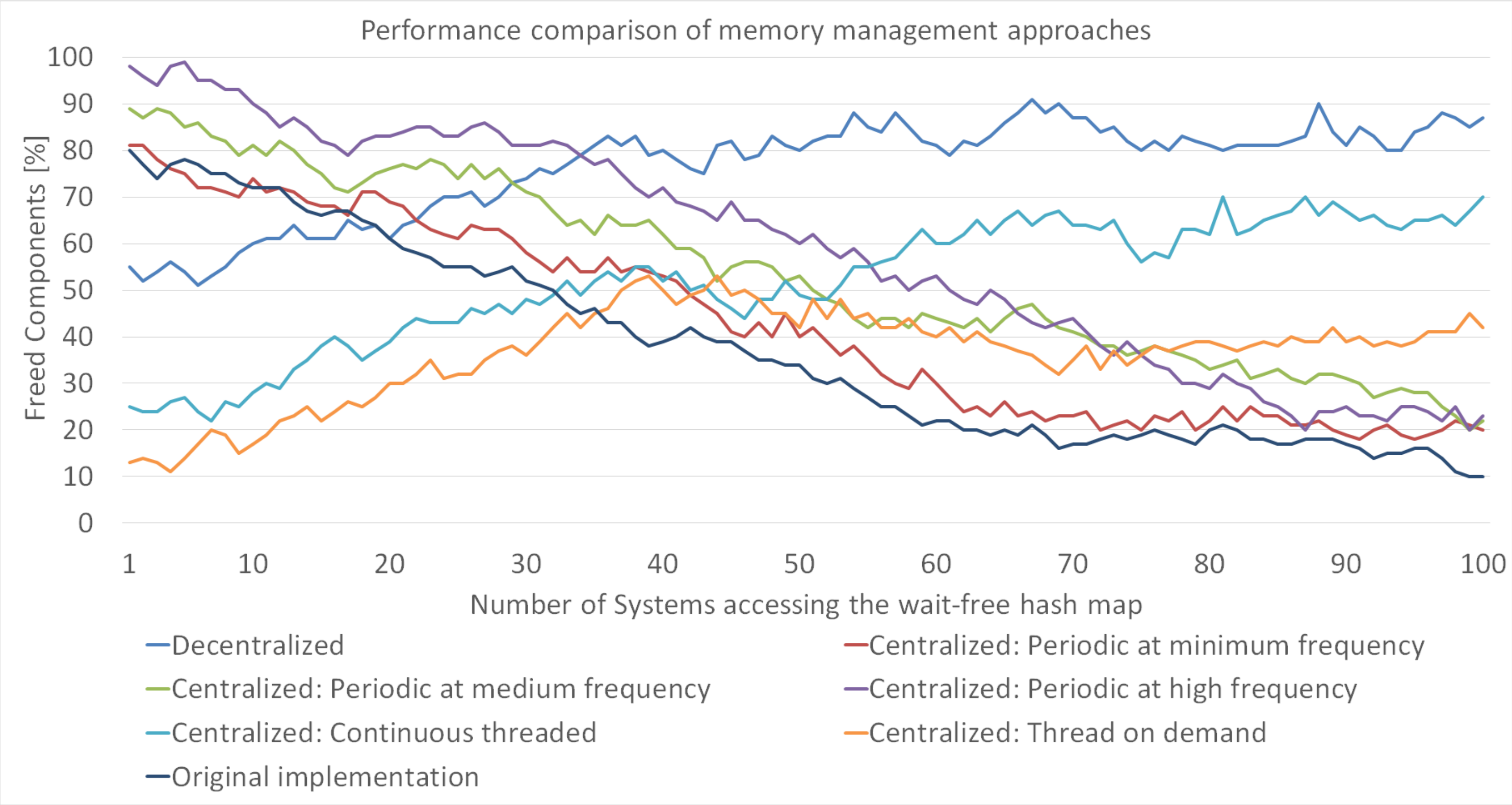
- Performance comparison of centralized and decentralized memory management implementations to original implementation
- Performance comparison of lock-based and wait-free hash map implementation

- Test configuration: Spaceflight mission simulator KaNaRiA
  - C++ with -O3 optimization
  - Each test averages 10,000 read/write operations with varying Component types (vectors, matrices, pointcloud data, strings, numerals)

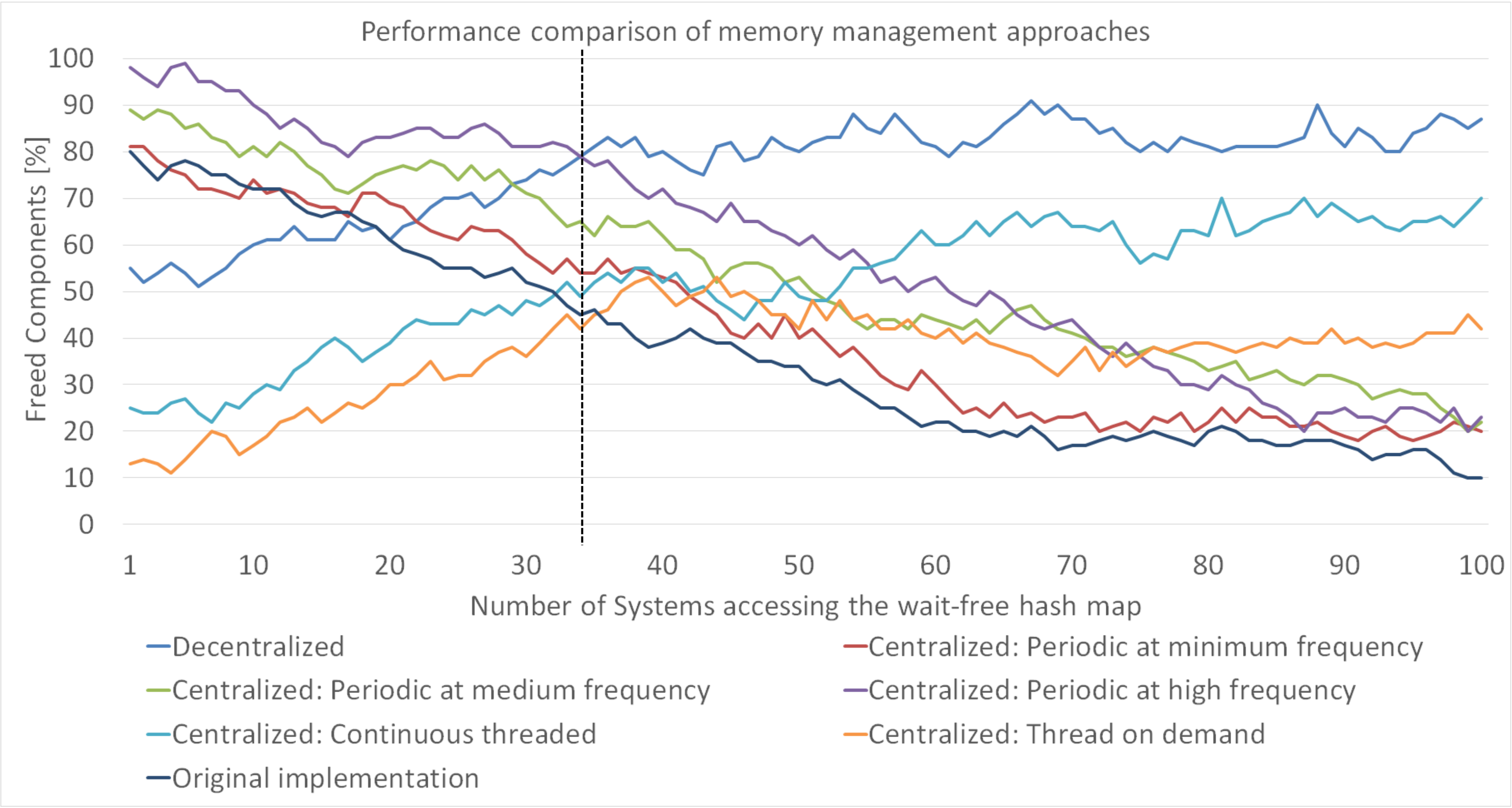
Results: Access Performance



Results: Memory Management



Results: Memory Management



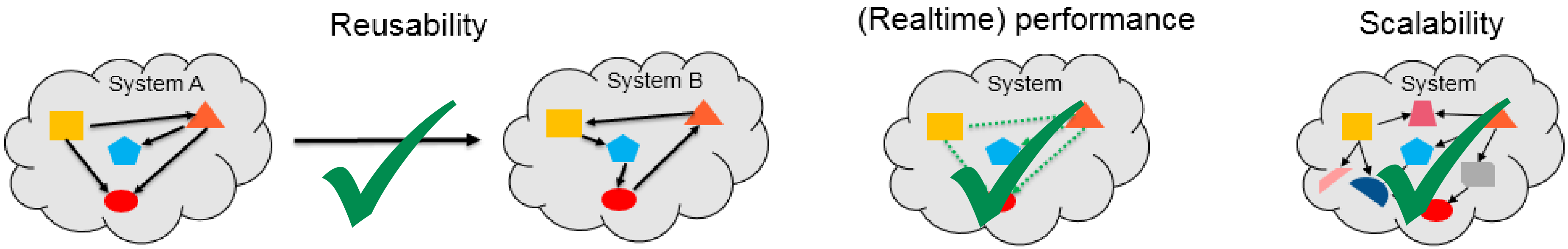
Best Practices

Few Systems	
Small Component data	Big Component data
Centralized (periodic with any frequency) management	Centralized (periodic with high frequency) management

Many Systems	
Small Component data	Big Component data
Decentralized management	Decentralized management

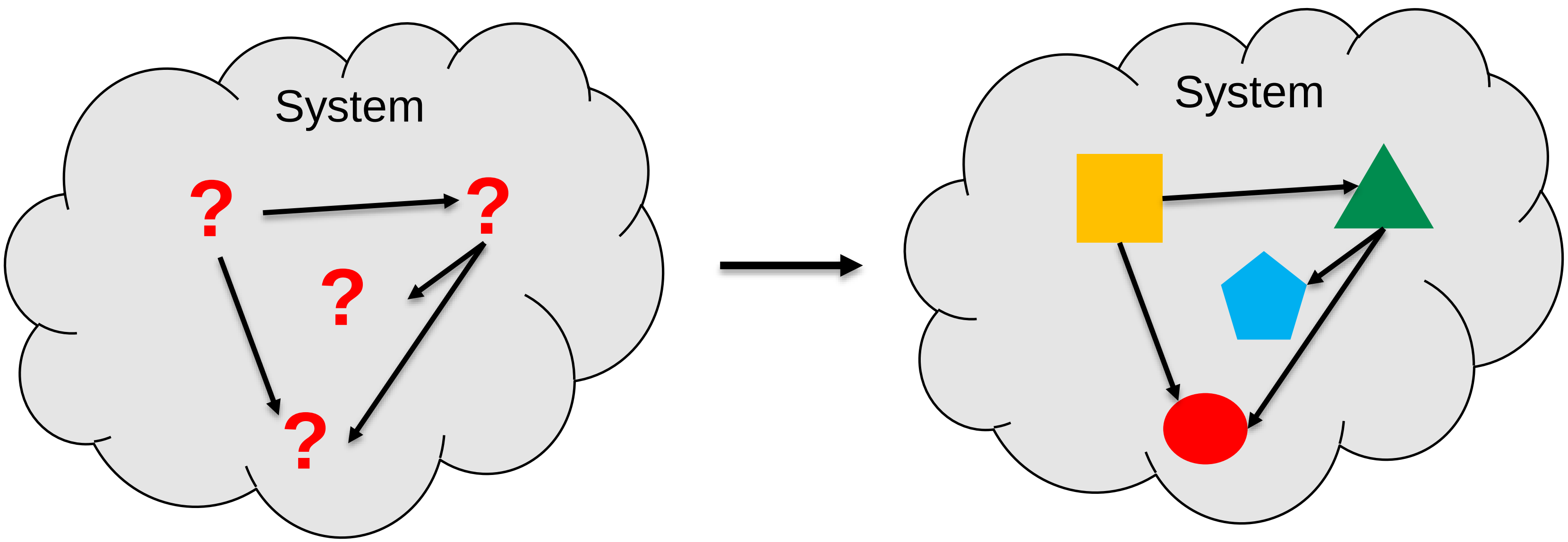
Our Contribution

- Novel extension of the ECS pattern for high performance double-buffered wait-free hash maps
  - Allows non-locking read and write operations
  - Highly responsive low-latency *Component* access for any number of *Systems*
- Novel efficient centralized and decentralized memory management for double-buffered wait-free hash maps
  - Reduces their memory consumption greatly by more than a factor of 10 while maintaining their high-performance access



Future Work

- High-level concepts for adaptive memory management
  - Determine current composition of ECS architecture
  - Autonomous switch between centralized and decentralized memory management



Thank you for your attention

Questions?

Patrick Lange, Rene Weller, Gabriel Zachmann
{lange,weller,zach}@cs.uni-bremen.de

This research is based upon the project KaNaRiA, supported by German Aerospace Center (DLR) with funds of German Federal Ministry of Economics and Technoloy (BMWi) grant *50NA1318*