C++ now

# Backporting Safety

Taylor Foxhall

2024

# Backporting Safety

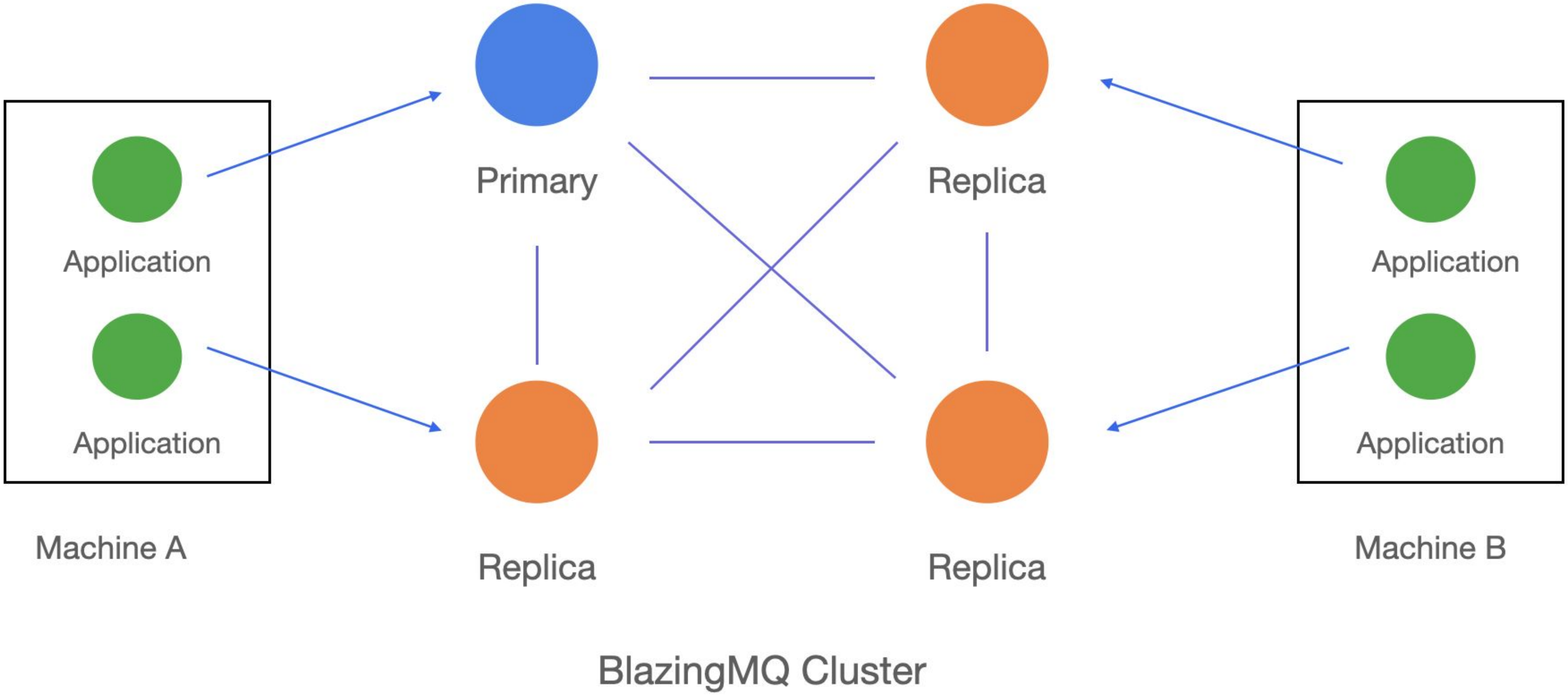
C++Now 2024
May 1, 2024

Taylor Foxhall
Bloomberg Managed Services (BMS) - Queuing Core

TechAtBloomberg.com

# BLAZINGMQ

- Distributed message queue
- Resilient to errors (“highly available”)
- Deployed in multi-hop topologies
- Published as open source in July 2023



Application

Application

Machine A

Primary

Replica

Replica

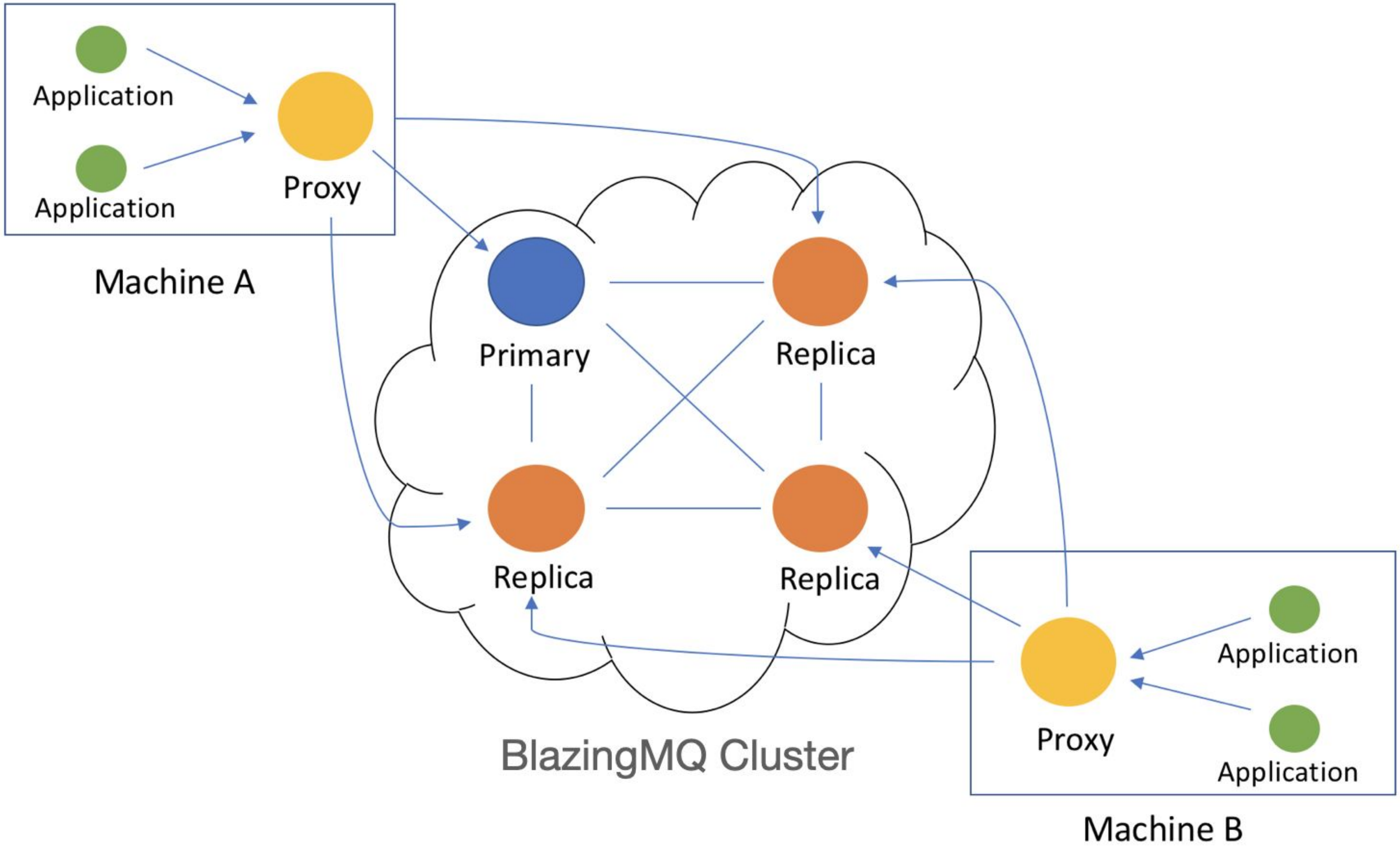
Replica

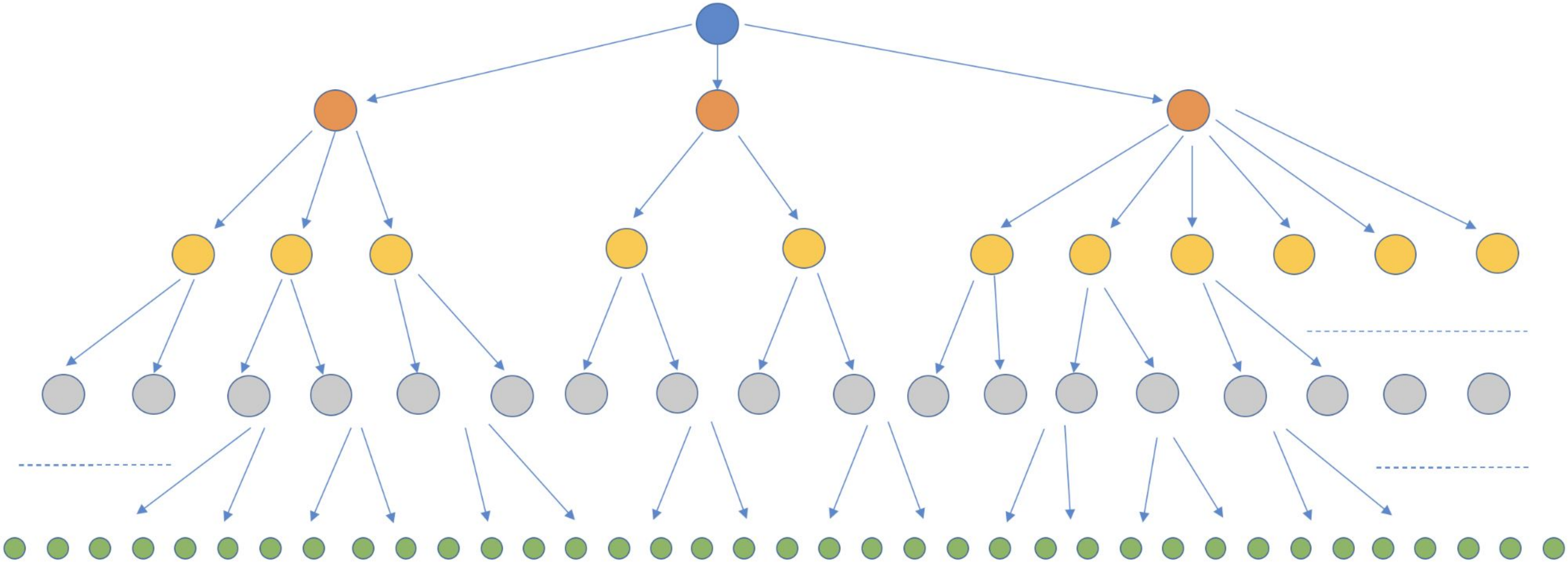
Application

Application

Machine B

BlazingMQ Cluster


Application
Application
Proxy
Machine A
Primary
Replica
Replica
Replica
BlazingMQ Cluster
Proxy
Application
Application
Machine B



# BLAZINGMQ

- Built on long existing C++ libraries
- Needed to meet performance goals
- Required support for platforms without C++11 compilers
- Resilient to unexpected states

# What is Safety?

- Memory safety
- Type safety
- Thread safety
- … safety

Keynote: Safety and Security: The Future of C++ - JF Bastien - CppNow 2023
https://www.youtube.com/watch?v=Gh79wcGJdTg

# What is Safety?

- “An operation is safe if it cannot lead to undefined behavior.”
- “An unsafe operation *may* lead to undefined behavior if its preconditions are violated.”

Sean Parent

Keynote: The Tragedy of C++, Acts One & Two - Sean Parent - CppNorth 2022
https://www.youtube.com/watch?v=kZCPURMH744

# Why Safety?

- Unsafe code can lead to undefined behavior
- Undefined behavior can cause *incorrect behavior*
- Some of that incorrect behavior is dangerous
- Many operations in C++ can cause undefined behavior by default
- C++ is trying to evolve to make it more difficult to do by default

https://herbsutter.com/2024/03/11/safety-in-context/

# Goals

- Explore causes for different types of safety bugs
- Highlight defensive design patterns techniques used to make BlazingMQ
- Show what possible defaults we can “backport” from future standards

# Memory Safety

TechAtBloomberg.com

Bloomberg

Engineering

# What is Memory Safety?

- Reading/writing out of bounds
- Use-after-free
- Using uninitialized data

# What is Memory Safety?

- Reading/writing out of bounds

Spatial

- Use-after-free
- Using uninitialized data

Temporal

The Meaning of Memory Safety https://arxiv.org/pdf/1705.07354.pdf

# Accepting Undefined Behaviors

- Undefined behavior is fundamental to C++ Standards
- We don’t have complete solutions
- Can we embrace them?

# Patching Memory Safety

| | |
|---|---|
| No out of bounds indexing | |
| No use-after-free | |
| No accessing uninitialized memory | |

# Patching Memory Safety

| | |
|---|---|
| **No out of bounds indexing** | |
| No use-after-free | |
| No accessing uninitialized memory | |

```
/// Returns the message corresponding to
/// id in the event queue.
int EventQueue::getMessage(size_t id) {
  return d_messages[id];
}
```

```
/// Returns the message corresponding to
/// id in the event queue.
int EventQueue::getMessage(size_t id) {
  return d_messages[id];
}
```

```
/// Returns the message corresponding to
/// id in the event queue.
/// @pre This function is undefined
/// unless 0 <= id < this->size()
int EventQueue::getMessage(size_t id) {
  return d_messages[id];
}
```

```
/// Returns the message corresponding to
/// id in the event queue.
/// @pre This function is undefined
/// unless 0 <= id < this->size()
int EventQueue::getMessage(size_t id) {
  CONTRACT_ASSERT(0 <= id
                  && id < size());
  return d_messages[id];
}
```

# Contract Programming in Brief

- Preconditions ⇒ checks before a function executes
- Postconditions ⇒ checks after a function returns
- Assertions ⇒ checks for everything else

# Patching Memory Safety

| | |
|---|---|
| **No out of bounds indexing** | Contract programming |
| No use-after-free | |
| No accessing uninitialized memory | |

# Patching Memory Safety

| | |
|---|---|
| No out of bounds indexing | Contract checking |
| **No use-after-free** | |
| No accessing uninitialized memory | |

```
/// Get the latest event.
const Event& EventQueue::getEvent() {
  return d_events.front();
}
```

# Use-After-Free

- How long does the return value of `getEvent()` live?
- Can programmers validate whether an `Event` is alive?
- What responsibility does `getEvent()` have to clients who still may store references to its return value?

```
/// Get the latest event.
shared_ptr<const Event>
EventQueue::getEvent() {
  // ...
}
```

# Garbage Collection

- Obviously the wrong default
- But it does solve the problem
- For some, it is the right default
- Not for BlazingMQ

# Patching Memory Safety

| | |
|---|---|
| No out of bounds indexing | Contract checking |
| **No use-after-free** | Garbage collection |
| No accessing uninitialized memory | |

# Patching Memory Safety

| | |
|---|---|
| No out of bounds indexing | Contract checking |
| **No use-after-free** | Garbage collection<br>+ ?? |
| No accessing uninitialized memory | |

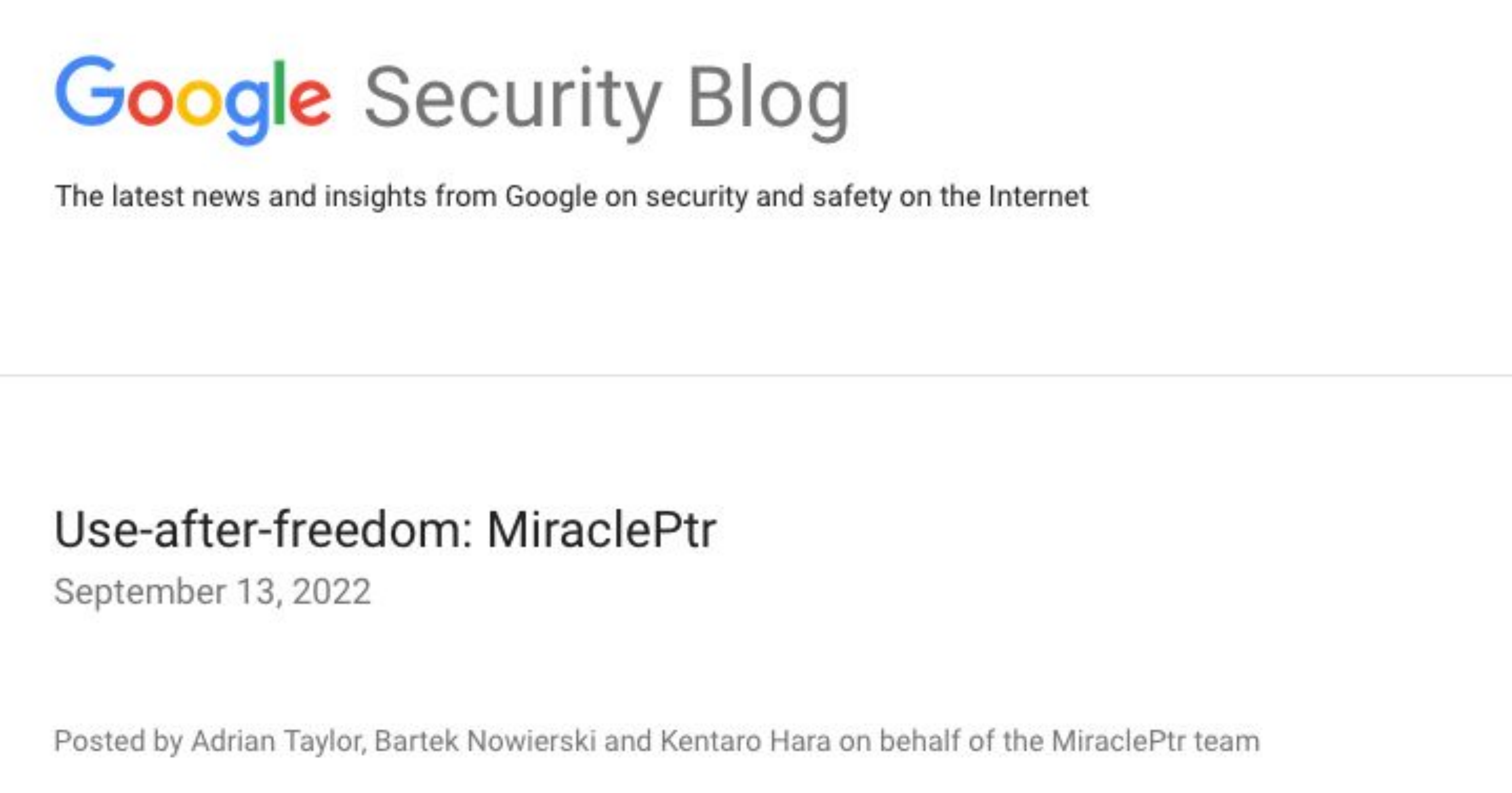
## Custom Allocators

- Obviously the wrong default
- But it does solve the problem
- For some, it is the right default
- Right for BlazingMQ

# Custom Allocators

- BlazingMQ uses them for memory leak detection
- Lets us rig the allocator with contracts
- We can control behavior as a definite memory leak is detected
- Like a mini valgrind!

# <Speculation>



Google Security Blog

The latest news and insights from Google on security and safety on the Internet

## Use-after-freedom: MiraclePtr

September 13, 2022

Posted by Adrian Taylor, Bartek Nowierski and Kentaro Hara on behalf of the MiraclePtr team

https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html

# Allocators & Contracts

- MiraclePtr combines a custom allocator with a `raw_ptr<T>` type
- Quarantines & poisons memory based on ref counts
- Types like `raw_ptr<T>` give us an opportunity to add a contract through `operator->/operator*`

```
template <typename T>
T* raw_ptr<T>::operator->() {
    CONTRACT_ASSERT(isAlive(d_ptr));
    return d_ptr;
}
```

```
template <typename T>
T* live(T* ptr) {
    CONTRACT_ASSERT(isAlive(ptr));
    return ptr;
}


int* a;
std::cout << *live(a);
```

# Allocators & Contracts

- Dereferencing invalid pointers is undefined

- C++ implementations could choose to make invalid pointer derefs fail a contract check

- Is this a better default?

# </Speculation>

# Patching Memory Safety

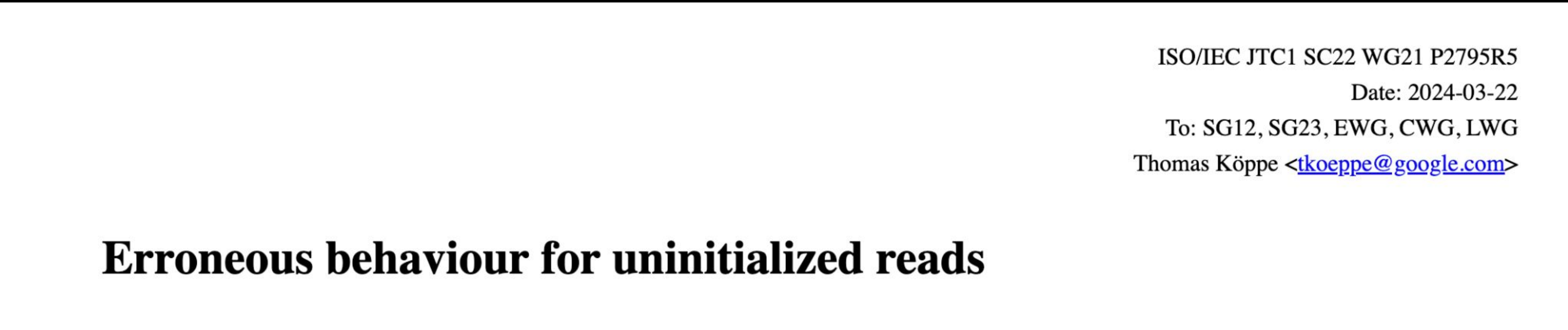
| | |
|---|---|
| No out of bounds indexing | Contract checking |
| **No use-after-free** | Garbage collection<br>Allocators & Contracts & raw_ptr? |
| No accessing uninitialized memory | |

# Patching Memory Safety

| | |
|---|---|
| No out of bounds indexing | Contract checking |
| No use-after-free | Garbage collection<br>Allocators & Contracts & raw_ptr? |
| **No accessing uninitialized memory** | |

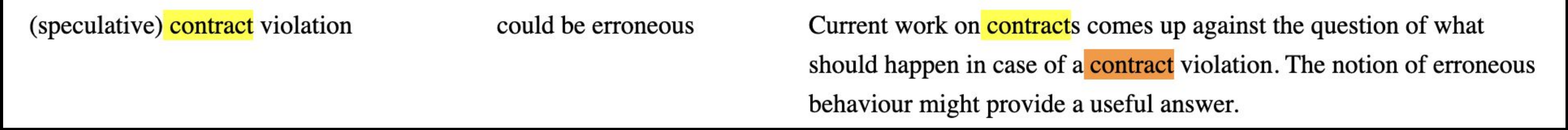
# Patching Memory Safety

| | |
|---|---|
| No out of bounds indexing | Contract checking |
| No use-after-free | Garbage collection<br>Allocators & Contracts & raw_ptr? |
| **No accessing uninitialized memory** | Constructors? |



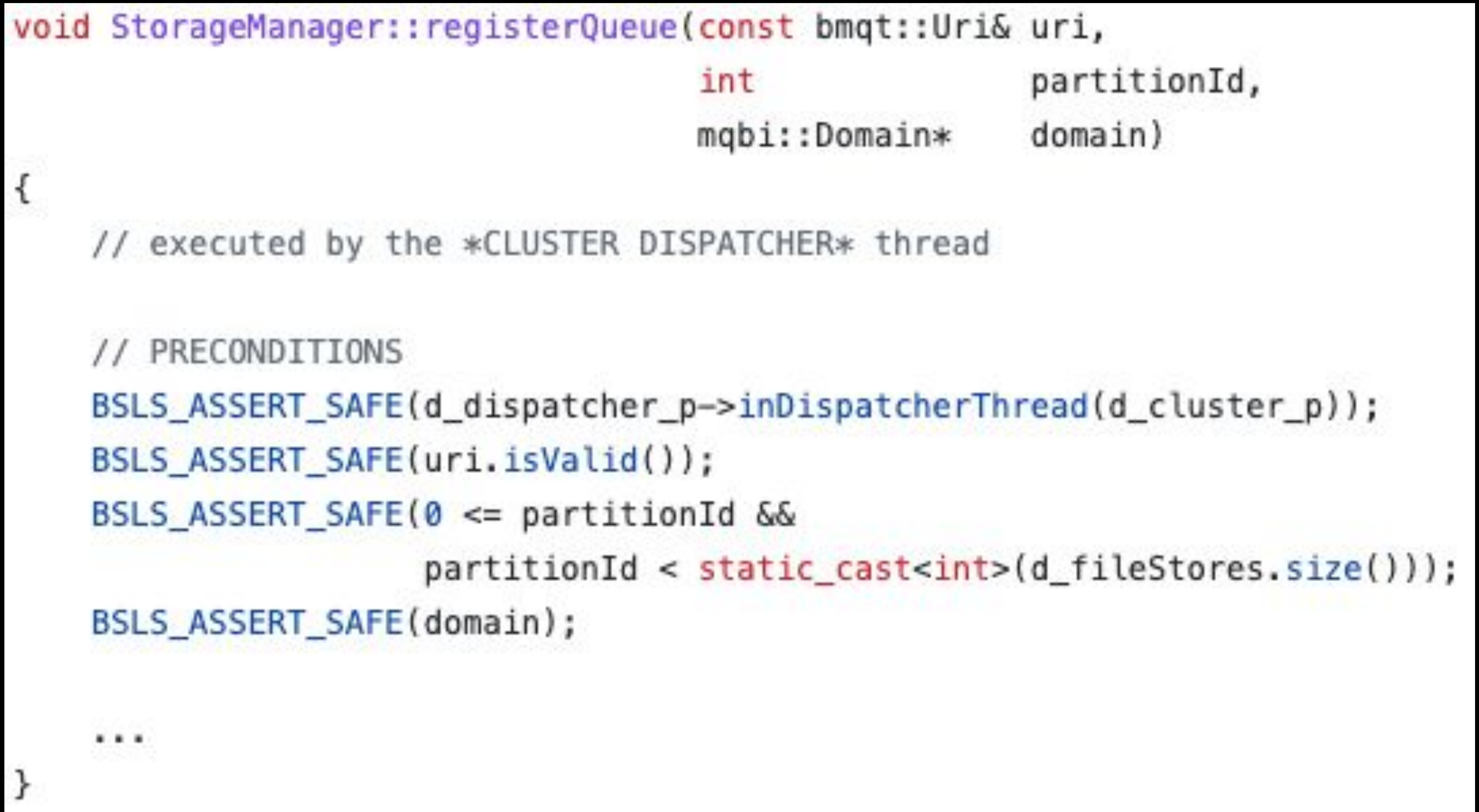
ISO/IEC JTC1 SC22 WG21 P2795R5
Date: 2024-03-22
To: SG12, SG23, EWG, CWG, LWG
Thomas Köppe <tkoeppe@google.com>

# Erroneous behaviour for uninitialized reads

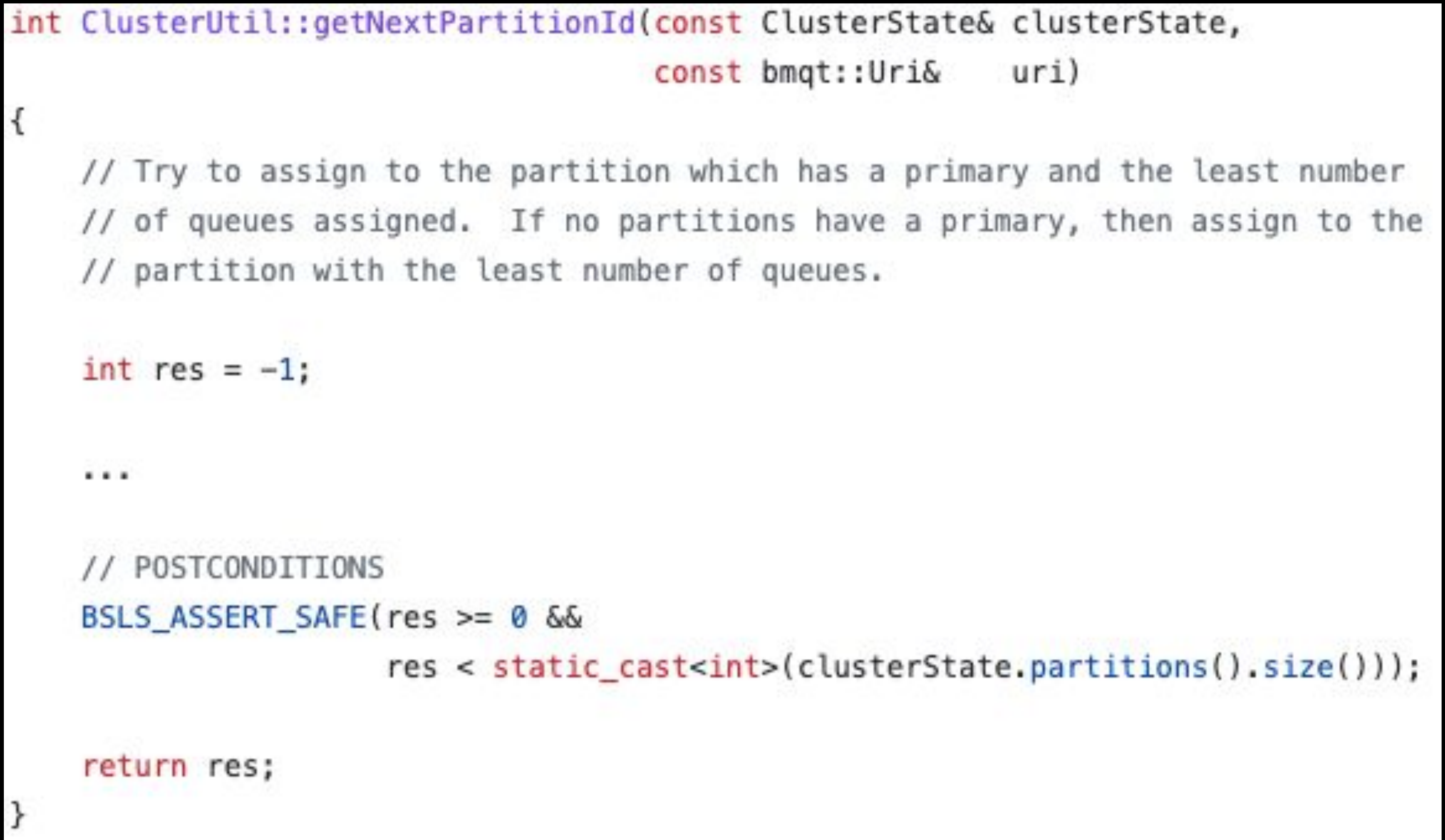


| | | |
|---|---|---|
| (speculative) contract violation | could be erroneous | Current work on contracts comes up against the question of what should happen in case of a contract violation. The notion of erroneous behaviour might provide a useful answer. |

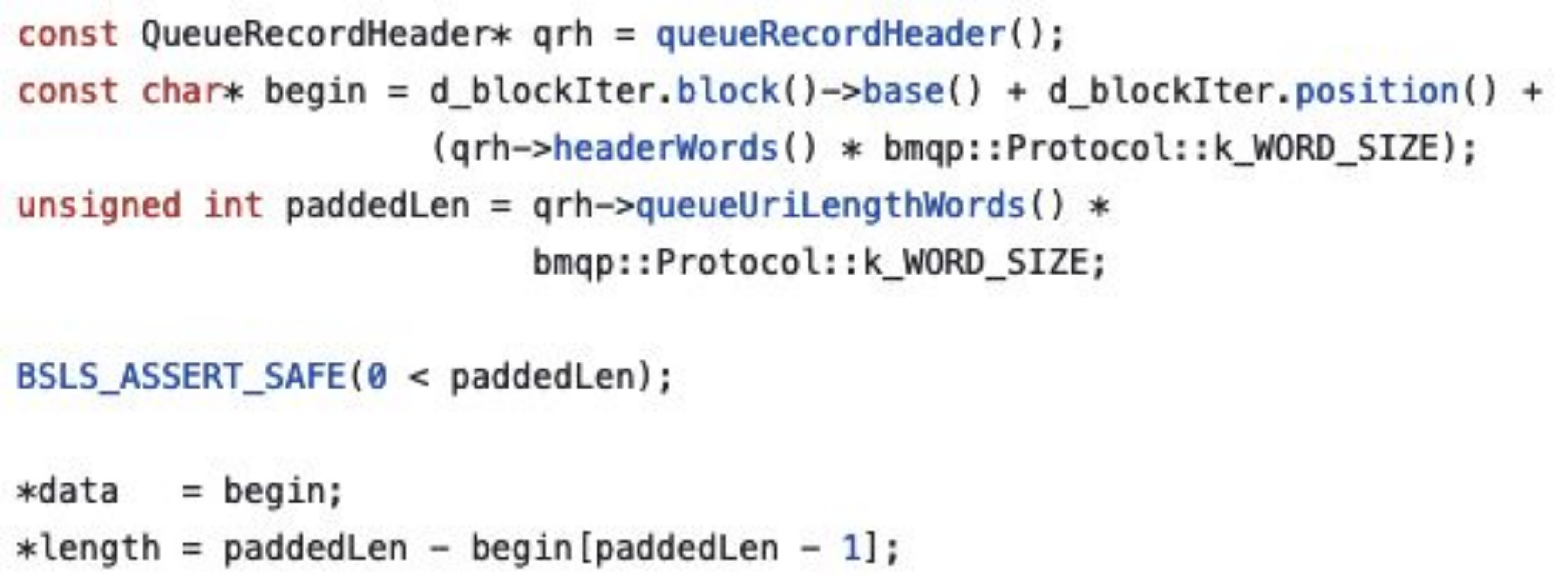
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html



```
void StorageManager::registerQueue(const bmqt::Uri& uri,
                                   int              partitionId,
                                   mqbi::Domain*    domain)
{
    // executed by the *CLUSTER DISPATCHER* thread

    // PRECONDITIONS
    BSLS_ASSERT_SAFE(d_dispatcher_p->inDispatcherThread(d_cluster_p));
    BSLS_ASSERT_SAFE(uri.isValid());
    BSLS_ASSERT_SAFE(0 <= partitionId &&
                     partitionId < static_cast<int>(d_fileStores.size()));
    BSLS_ASSERT_SAFE(domain);

    ...
}
```



```
int ClusterUtil::getNextPartitionId(const ClusterState& clusterState,
                                    const bmqt::Uri&    uri)
{
    // Try to assign to the partition which has a primary and the least number
    // of queues assigned.  If no partitions have a primary, then assign to the
    // partition with the least number of queues.

    int res = -1;

    ...

    // POSTCONDITIONS
    BSLS_ASSERT_SAFE(res >= 0 &&
                     res < static_cast<int>(clusterState.partitions().size()));

    return res;
}
```



```
const QueueRecordHeader* qrh = queueRecordHeader();
const char* begin = d_blockIter.block()->base() + d_blockIter.position() +
                    (qrh->headerWords() * bmqp::Protocol::k_WORD_SIZE);
unsigned int paddedLen = qrh->queueUriLengthWords() *
                         bmqp::Protocol::k_WORD_SIZE;

BSLS_ASSERT_SAFE(0 < paddedLen);

*data   = begin;
*length = paddedLen - begin[paddedLen - 1];
```

# Type Safety

TechAtBloomberg.com

Bloomberg

Engineering

# Type Safety

- Prevent invalid (perhaps undefined) operations on data
- “Make it impossible/hard to do the wrong thing”
- Examples
  - `std::optional/std::variant`
  - URIs
  - Message protocols

TechAtBloomberg.com

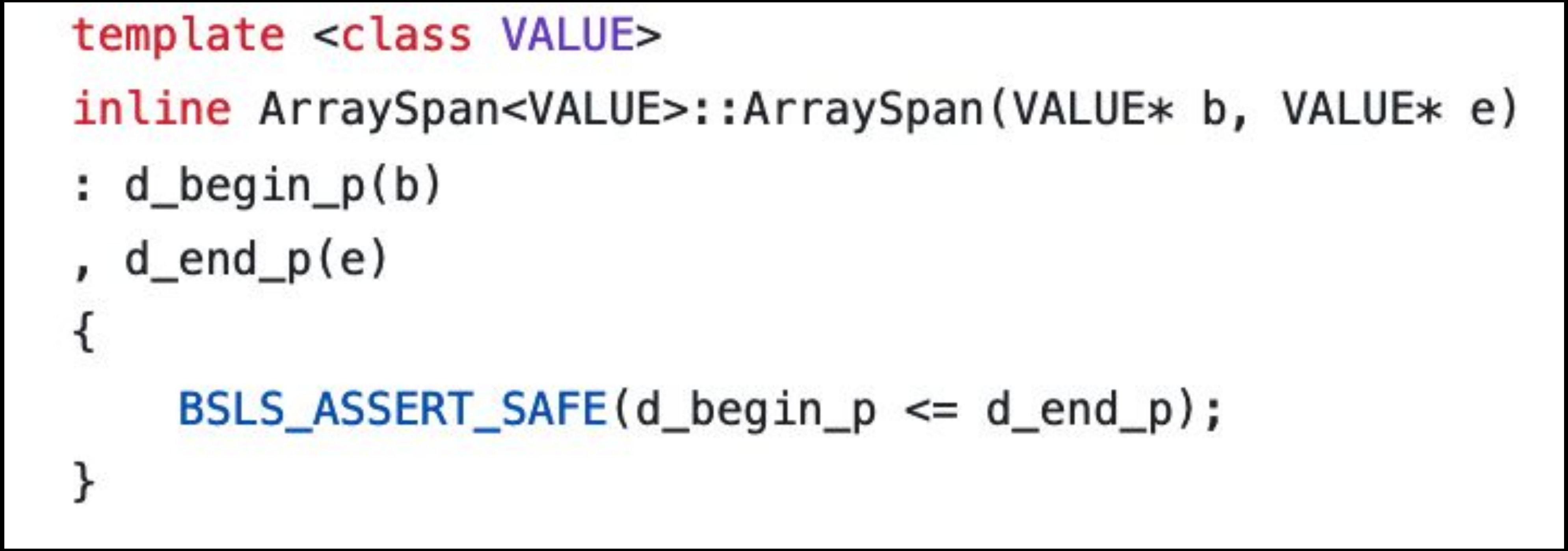
Bloomberg

Engineering

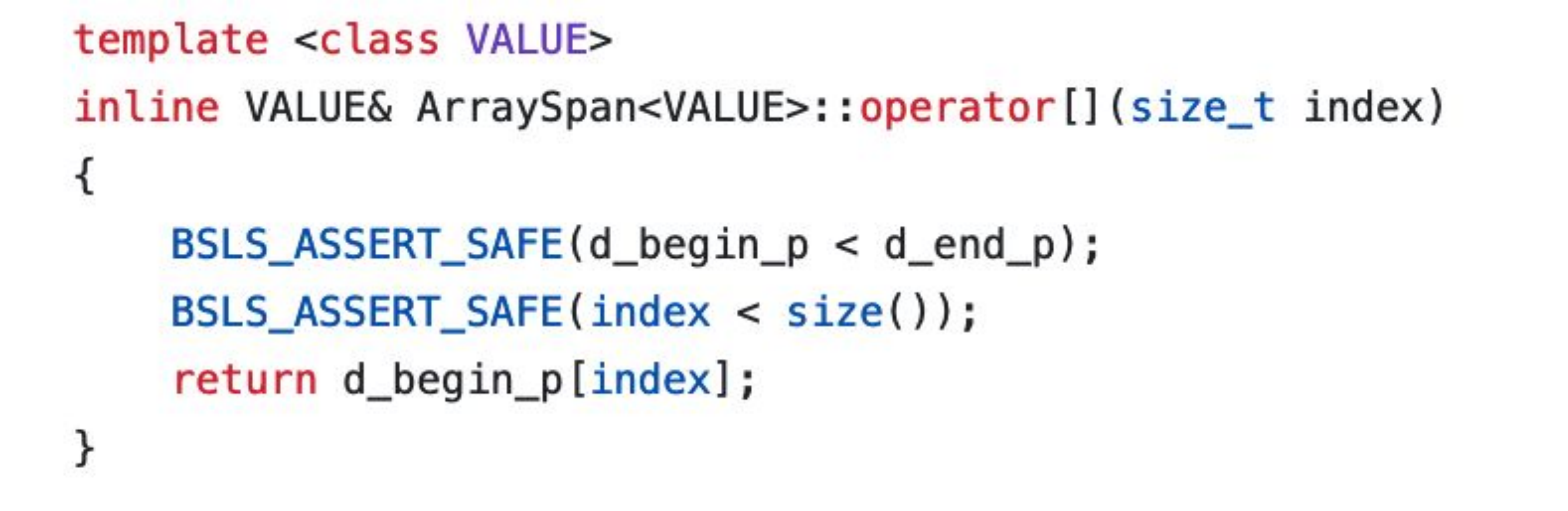
# Type Safety

- Part of the advantage of types was that some of those “wrong things” were other safety bugs!

- Many types from future standards are literally backportable
  - e.g. `optional`, `shared_ptr`, `unordered_map`, `array`



```
template <class VALUE>
inline ArraySpan<VALUE>::ArraySpan(VALUE* b, VALUE* e)
: d_begin_p(b)
, d_end_p(e)
{
    BSLS_ASSERT_SAFE(d_begin_p <= d_end_p);
}
```



```
template <class VALUE>
inline VALUE& ArraySpan<VALUE>::operator[](size_t index)
{
    BSLS_ASSERT_SAFE(d_begin_p < d_end_p);
    BSLS_ASSERT_SAFE(index < size());
    return d_begin_p[index];
}
```

# Thread Safety

TechAtBloomberg.com

Bloomberg

Engineering

# Data Races

- Data races are undefined behavior by the standard

- A data race is:
  - Two threads use data at the same time
  - At least one of them is modifying the data

- Can cause objects to be invalidated

- That might to lead to other downstream safety bugs!

# Ways to Fix Data Races

- Make everything const
- Synchronize access to data with atomics & mutexes
- Prevent sharing data

# Actor Model

- No sharing memory
- Computations are isolated into individual threads of execution
- Actors can pass messages to each other

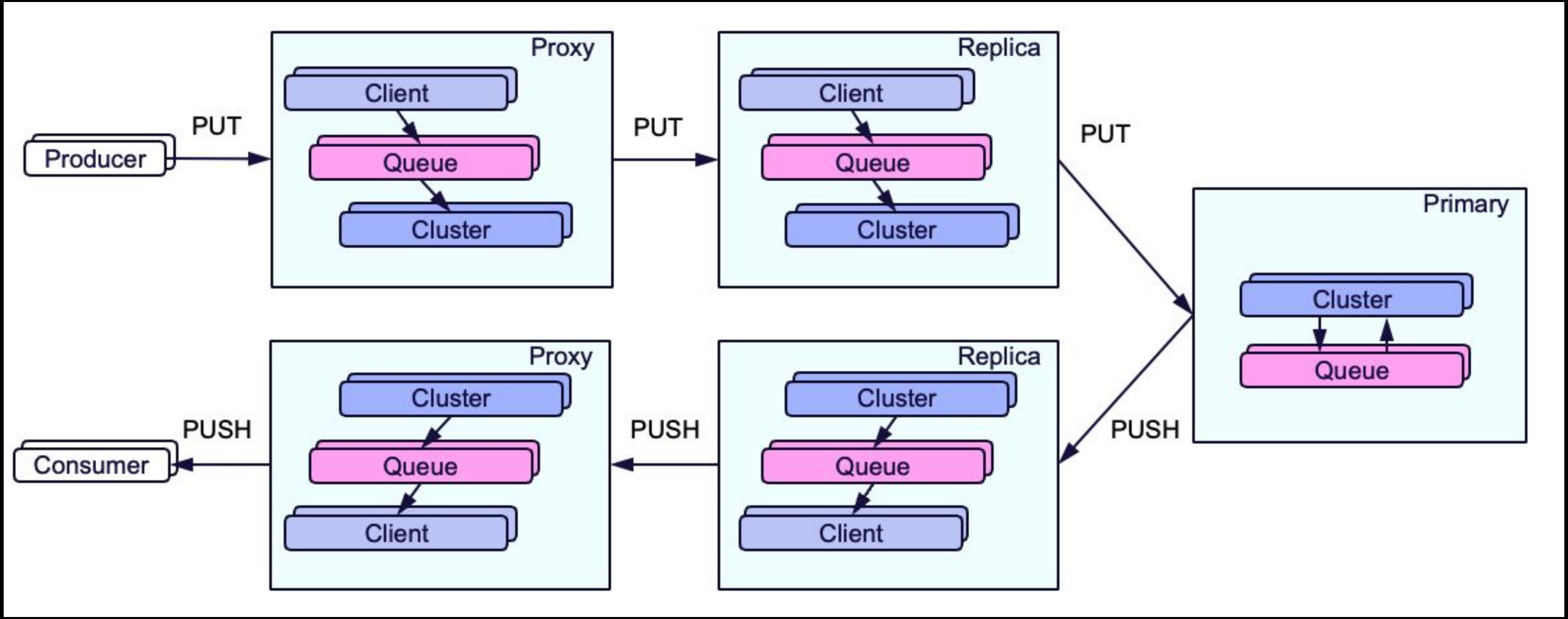
# Actor Model in BlazingMQ

- Client
  - Reading/writing to client
  - Stats and message validation

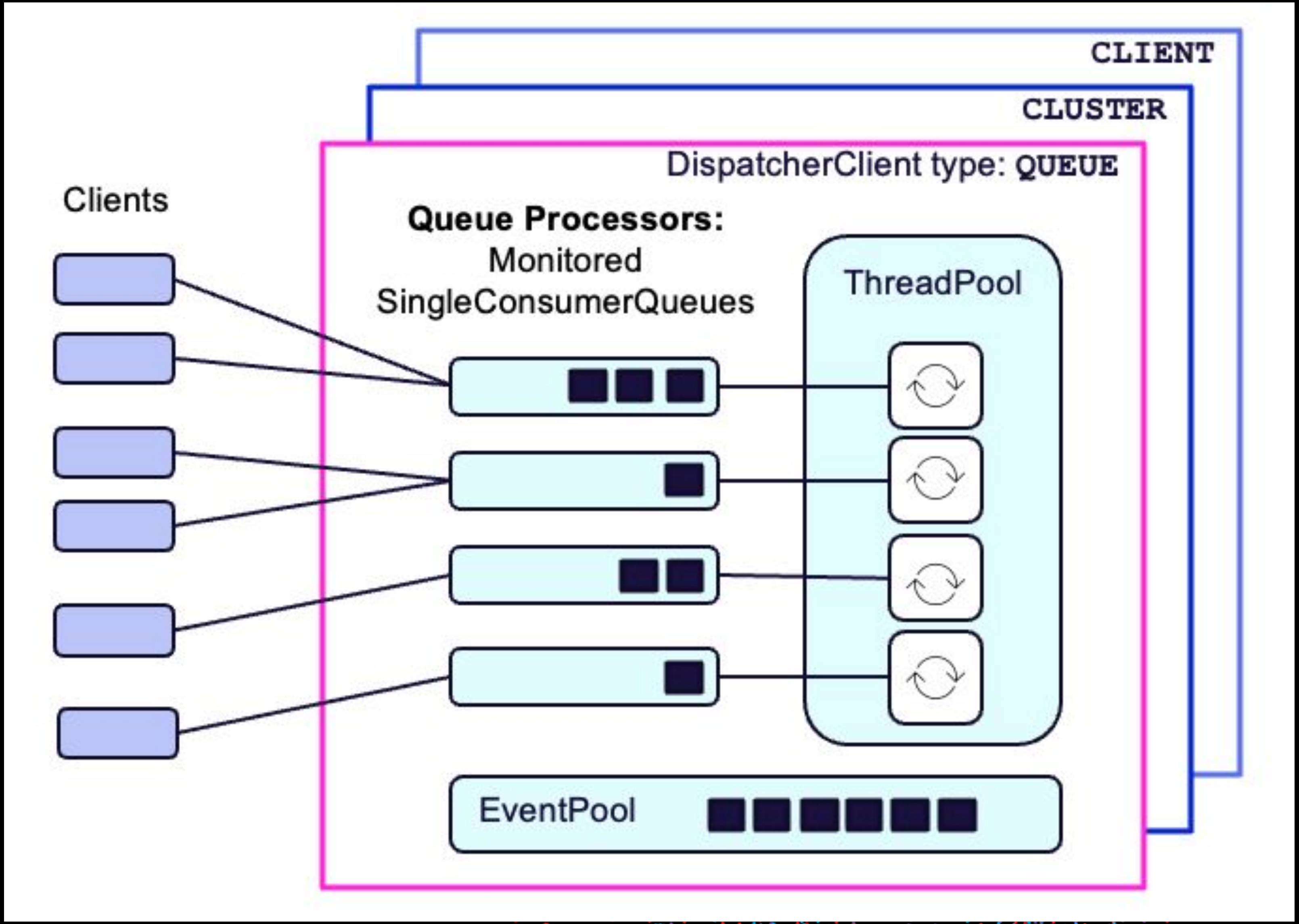
- Queue
  - Storage and replication
  - Data routing

- Cluster
  - Reading/writing to cluster nodes
  - Cluster health
  - Primary node

# Actor Model in BlazingMQ

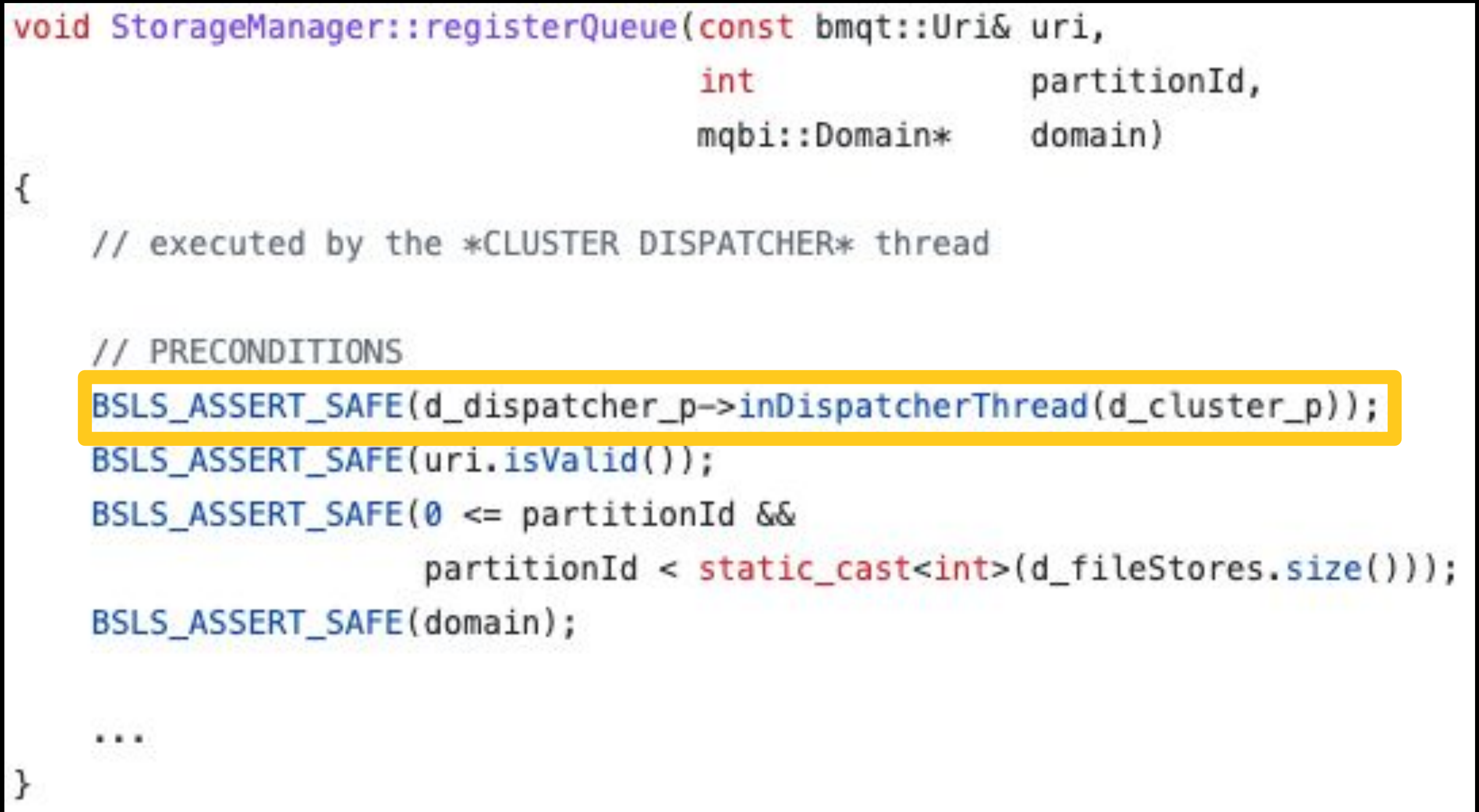


# Event Dispatcher



Bloomberg

Engineering



```
void StorageManager::registerQueue(const bmqt::Uri& uri,
                                   int              partitionId,
                                   mqbi::Domain*    domain)
{
    // executed by the *CLUSTER DISPATCHER* thread

    // PRECONDITIONS
    BSLS_ASSERT_SAFE(d_dispatcher_p->inDispatcherThread(d_cluster_p));
    BSLS_ASSERT_SAFE(uri.isValid());
    BSLS_ASSERT_SAFE(0 <= partitionId &&
                     partitionId < static_cast<int>(d_fileStores.size()));
    BSLS_ASSERT_SAFE(domain);


    ...
}
```

# Conclusion

TechAtBloomberg.com

Bloomberg

Engineering

# What Did We Learn?

- When talking about safety in C++, qualify definitions
- Types are more easily backported than language features
- Maybe we need some design patterns for safety
- Undefined behavior has utility

# Thank you!

https://techatbloomberg.com/cplusplus

https://www.bloomberg.com/careers

Bloomberg Engineering

TechAtBloomberg.com

# Postscript: Correctness

TechAtBloomberg.com

Bloomberg

Engineering

# Testing

- Unit Tests
- Integration Tests
- Sanitizers
  - ASAN, TSAN, MSAN, UBSAN
- System Correctness Verification

# System Testing with Jepsen

- Nemesis: network partitions, start/stop node, clock skews, etc.

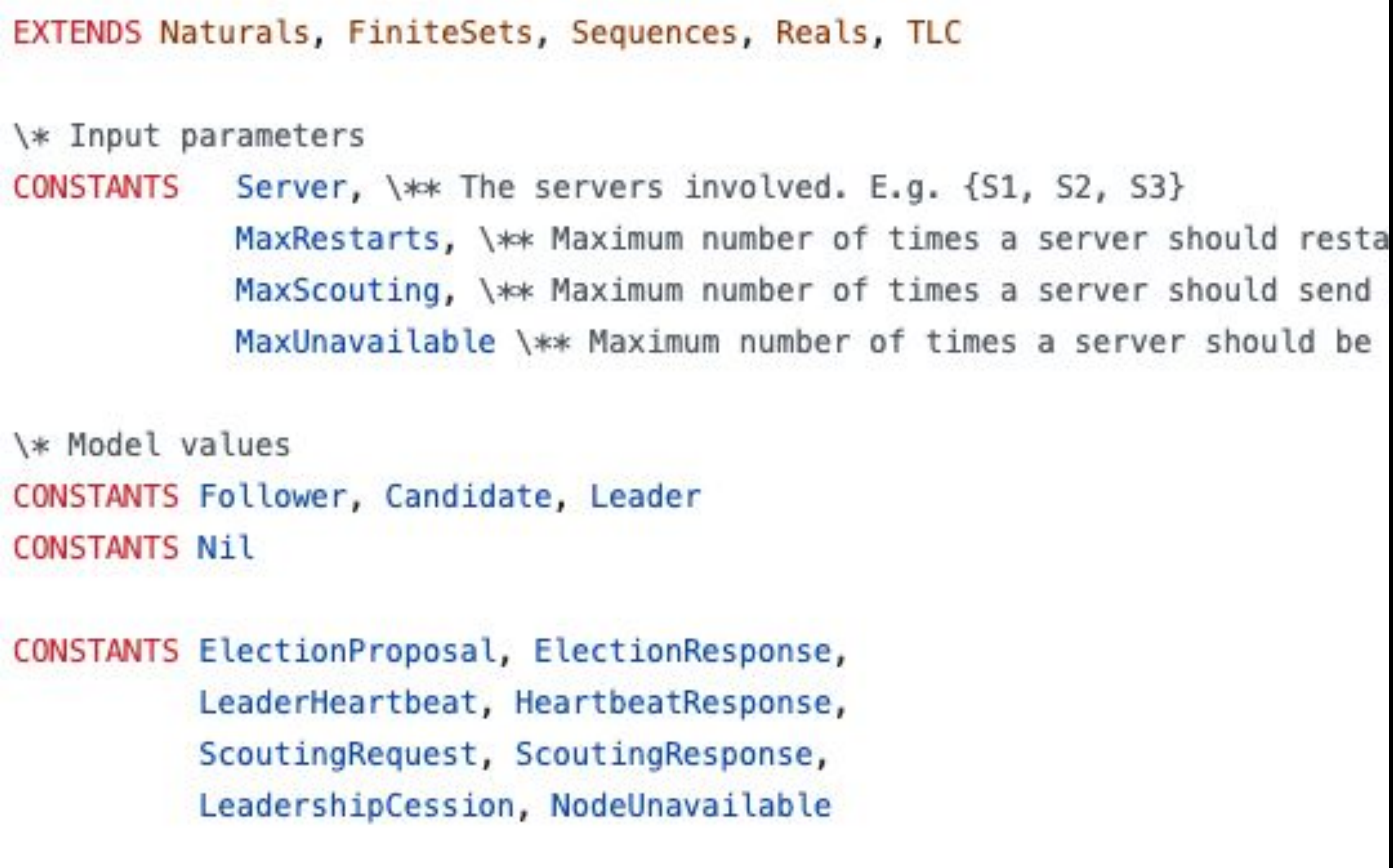
| | BlazingMQ | |
|---|---|---|
| | Eventual Consistency | Strong Consistency |
| partition-random-node | ✅ | ✅ |
| partition-random-halves | ❌ | ✅ |
| partition-majorities-ring | ❌ | ✅ |

# TLA+

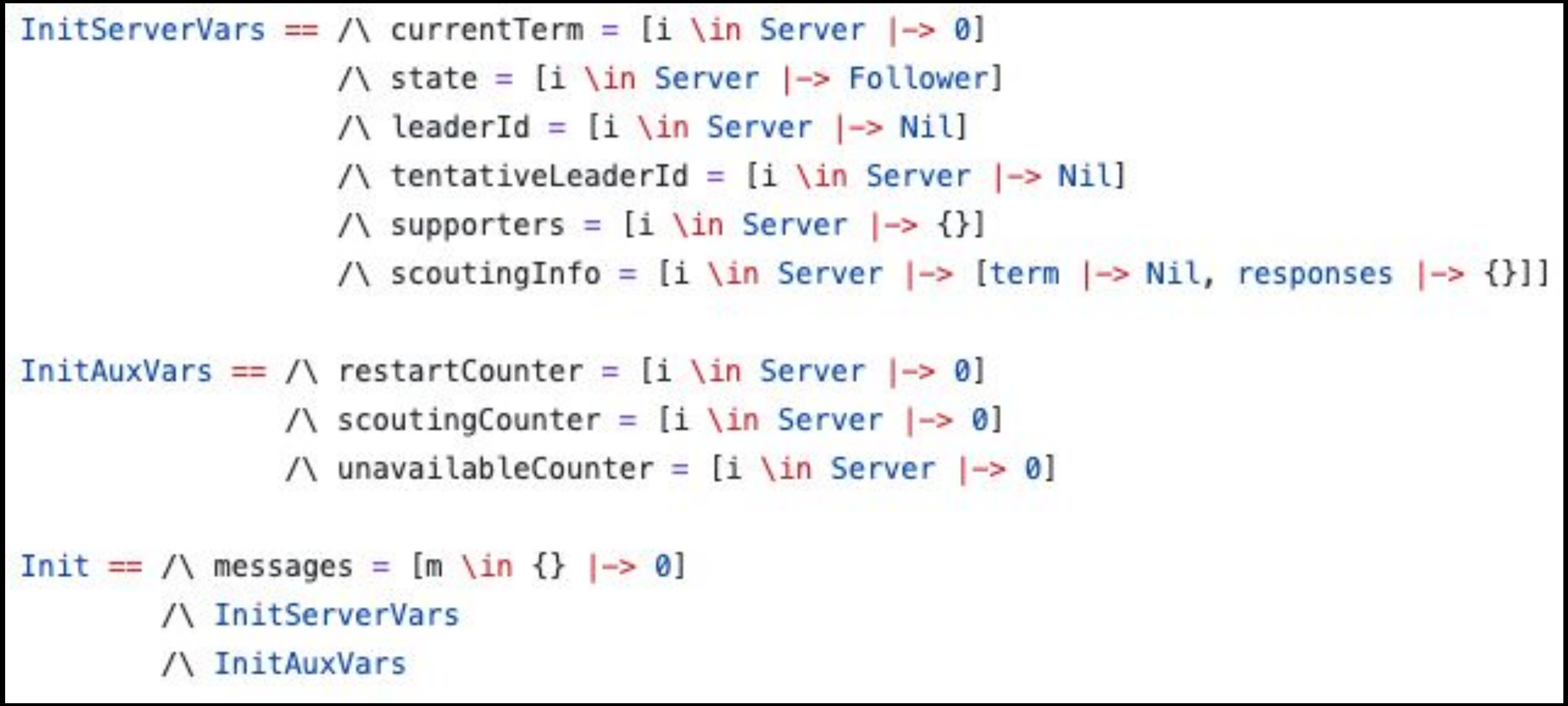
- Formal specification language

- BlazingMQ's leader election and state machine replication is strongly inspired by the Raft consensus algorithm

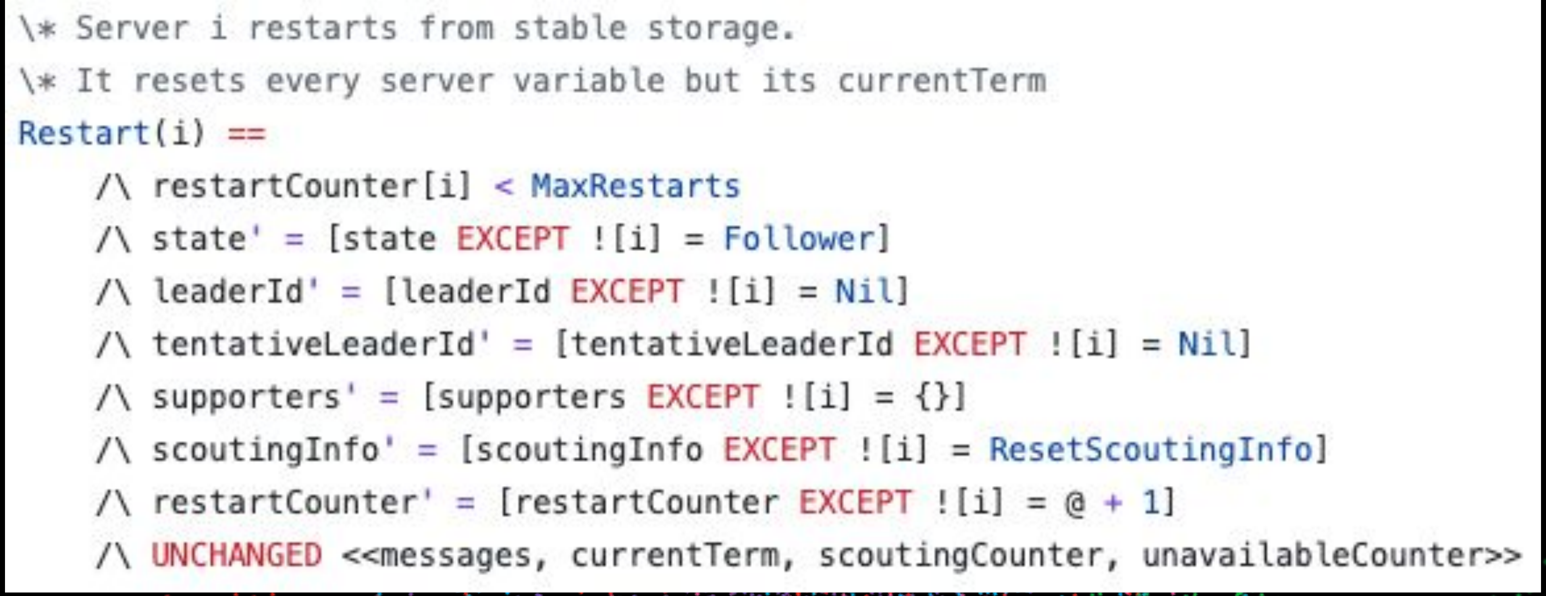
- TLA+ verifies the correctness of the implementation



```
EXTENDS Naturals, FiniteSets, Sequences, Reals, TLC

\* Input parameters
CONSTANTS   Server, \** The servers involved. E.g. {S1, S2, S3}
            MaxRestarts, \** Maximum number of times a server should resta
            MaxScouting, \** Maximum number of times a server should send
            MaxUnavailable \** Maximum number of times a server should be

\* Model values
CONSTANTS Follower, Candidate, Leader
CONSTANTS Nil

CONSTANTS ElectionProposal, ElectionResponse,
          LeaderHeartbeat, HeartbeatResponse,
          ScoutingRequest, ScoutingResponse,
          LeadershipCession, NodeUnavailable
```



```
InitServerVars == /\ currentTerm = [i \in Server |-> 0]
                  /\ state = [i \in Server |-> Follower]
                  /\ leaderId = [i \in Server |-> Nil]
                  /\ tentativeLeaderId = [i \in Server |-> Nil]
                  /\ supporters = [i \in Server |-> {}]
                  /\ scoutingInfo = [i \in Server |-> [term |-> Nil, responses |-> {}]]

InitAuxVars == /\ restartCounter = [i \in Server |-> 0]
               /\ scoutingCounter = [i \in Server |-> 0]
               /\ unavailableCounter = [i \in Server |-> 0]

Init == /\ messages = [m \in {} |-> 0]
        /\ InitServerVars
        /\ InitAuxVars
```



```
\* Server i restarts from stable storage.
\* It resets every server variable but its currentTerm
Restart(i) ==
    /\ restartCounter[i] < MaxRestarts
    /\ state' = [state EXCEPT ![i] = Follower]
    /\ leaderId' = [leaderId EXCEPT ![i] = Nil]
    /\ tentativeLeaderId' = [tentativeLeaderId EXCEPT ![i] = Nil]
    /\ supporters' = [supporters EXCEPT ![i] = {}]
    /\ scoutingInfo' = [scoutingInfo EXCEPT ![i] = ResetScoutingInfo]
    /\ restartCounter' = [restartCounter EXCEPT ![i] = @ + 1]
    /\ UNCHANGED <<messages, currentTerm, scoutingCounter, unavailableCounter>>
```

# References

- BlazingMQ: https://github.com/bloomberg/blazingmq
- BlazingMQ landing page: https://bloomberg.github.io/blazingmq/

- BDE: https://github.com/bloomberg/bde
- Actor Model: https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf
- Jepsen: https://jepsen.io/
- TLA+: https://lamport.azurewebsites.net/tla/tla.html

- Our P99 CONF talk: https://www.p99conf.io/session/architecting-a-high-performance-open-source-distributed-message-queuing-system-in-c/

# References

- The Meaning of Memory Safety: https://arxiv.org/pdf/1705.07354.pdf
- The Urgent Need for Memory Safety in Software Products: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products
- The Case for Memory Safe Roadmaps: https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf

# References

- Safety and Security: The Future of C++ - JF Bastien - CppNow 2023
  https://www.youtube.com/watch?v=Gh79wcGJdTg

- All the Safeties: Safety in C++ - Sean Parent - CppNow 2023
  https://www.youtube.com/watch?v=MO-qehjc04s

- Delivering Safe C++ - Bjarne Stroustrup - CppCon 2023
  https://www.youtube.com/watch?v=I8UvQKvOSSw