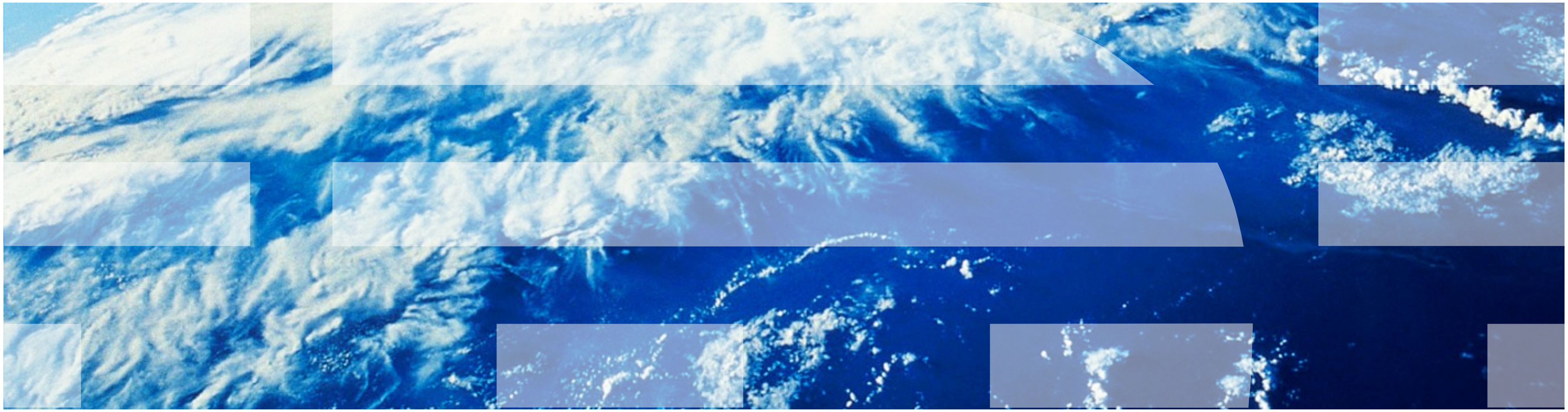
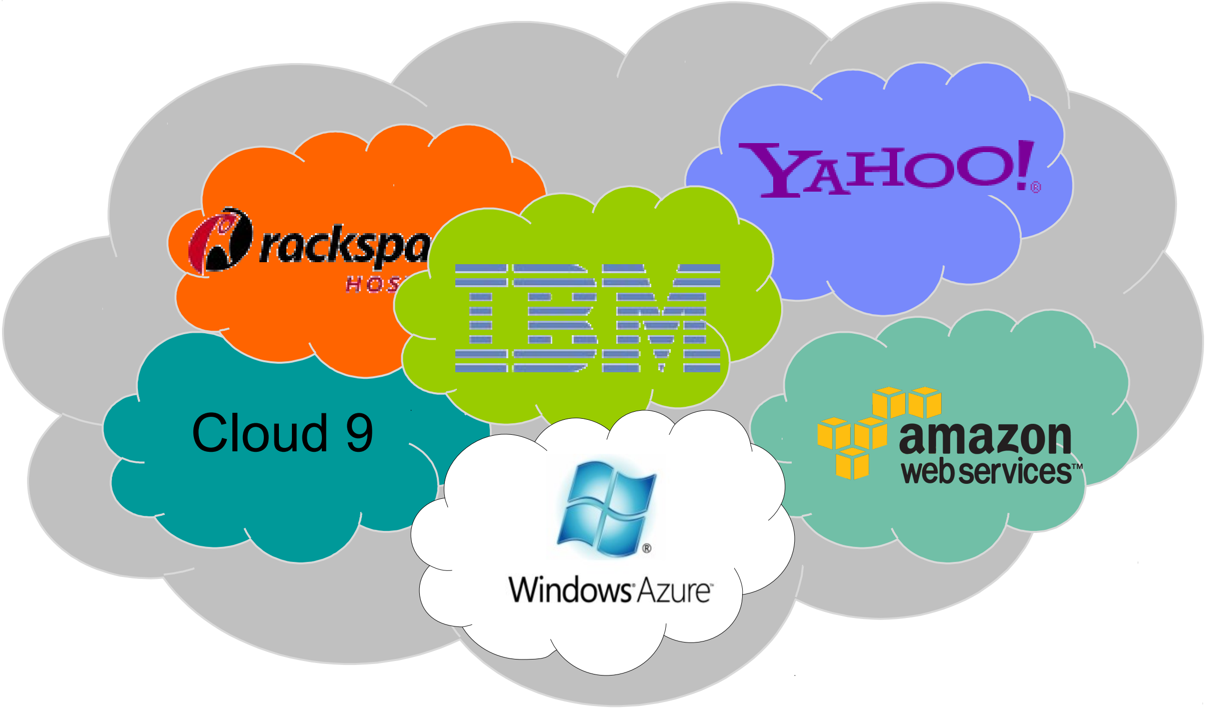
Alessandro Sorniotti

Cloud and intercloud Storage



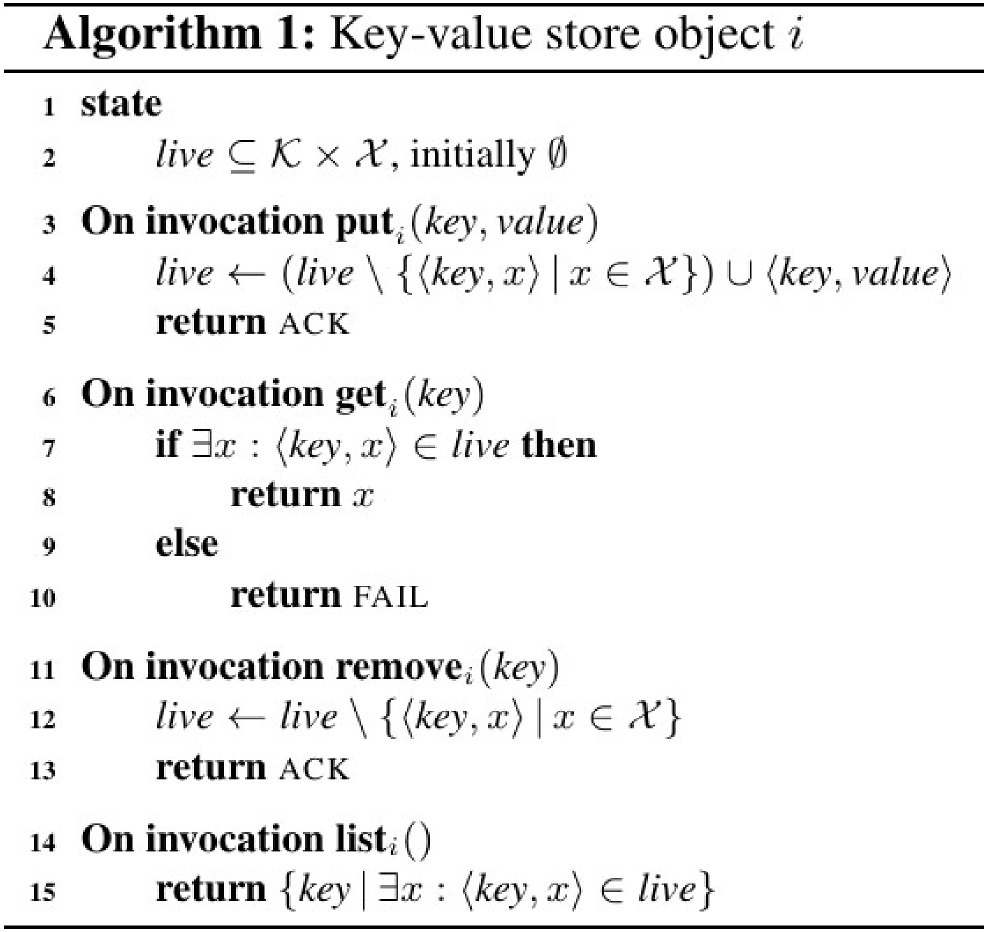
Overview

From Cloud to Intercloud



- **Cloud-based object storage systems is a success story**
    - – Prices and scale which can't be met with traditional architectures
    - – Popular and successful (Amazon S3 already stores 762 billion objects)
    - – Simple APIs (KVS)
- Cheap
- Simplicity however goes hand in hand with **lack of enterprise features**

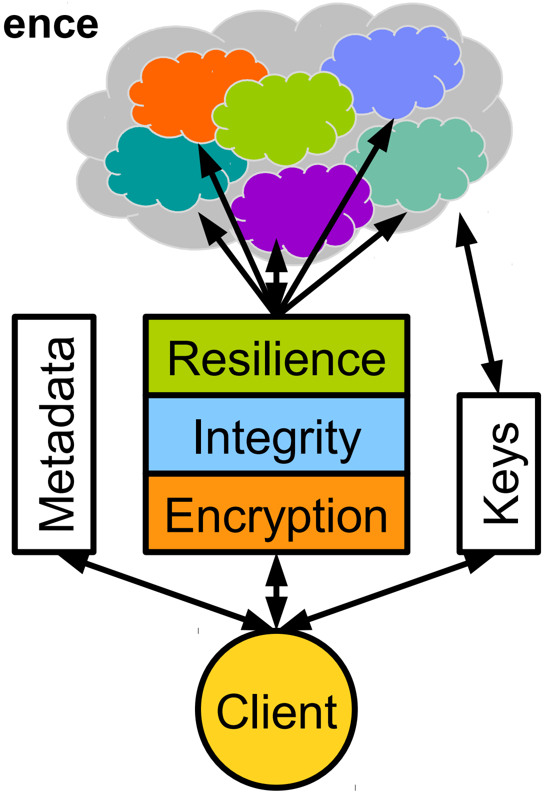
KVS



```
Algorithm 1: Key-value store object $i$
state
    $live \subseteq \mathcal{K} \times \mathcal{X}$, initially $\emptyset$
On invocation put$_i(key, value)$
    $live \leftarrow (live \setminus \{\langle key, x \rangle \mid x \in \mathcal{X}\}) \cup \langle key, value \rangle$
    return ACK
On invocation get$_i(key)$
    if $\exists x : \langle key, x \rangle \in live$ then
        return $x$
    else
        return FAIL
On invocation remove$_i(key)$
    $live \leftarrow live \setminus \{\langle key, x \rangle \mid x \in \mathcal{X}\}$
    return ACK
On invocation list$_i()$
    return $\{key \mid \exists x : \langle key, x \rangle \in live\}$
```

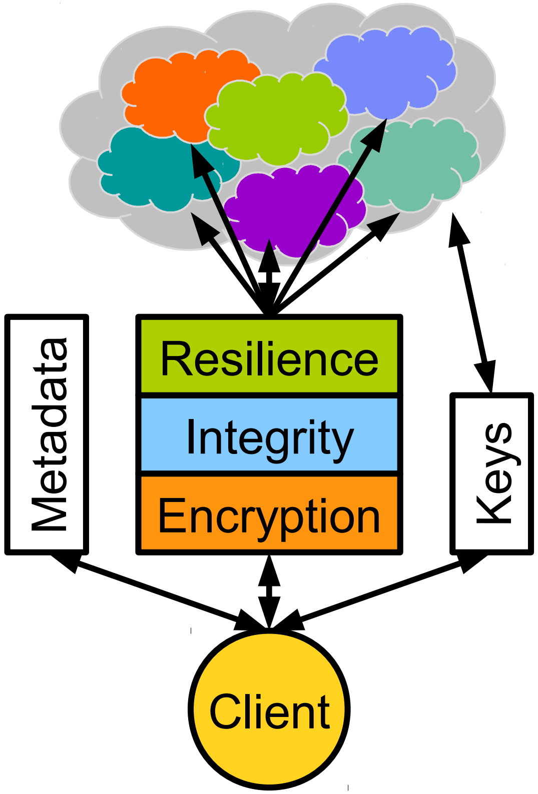
ICStore

- A Java-based **intercloud storage client** developed in ZRL
  - Can “talk” to 20+ (and counting) KVS-based storage services
  - No modification to remote clouds
- **Modular**, **layered** library, offering **encryption**, **integrity** and **resilience**
  - Layers are configurable and switchable
  - Lightweight, **asynchronous**, **multi-threaded** architecture
  - **Streamed** object operations (no buffering required)
  - **Buffering** of unsuccessful operations for “slow” clouds
- **Transparent** to client (e.g. proxy, gw)
- Exposes KVS APIs
  - De-facto standard for “web 2.0” data storage
  - Easy interoperability with existing applications/appliances
  - May be turned into file-based storage (e.g. using s3fs)
- Can **scale up** or **scale down** very easily
  - No client-to-client communication



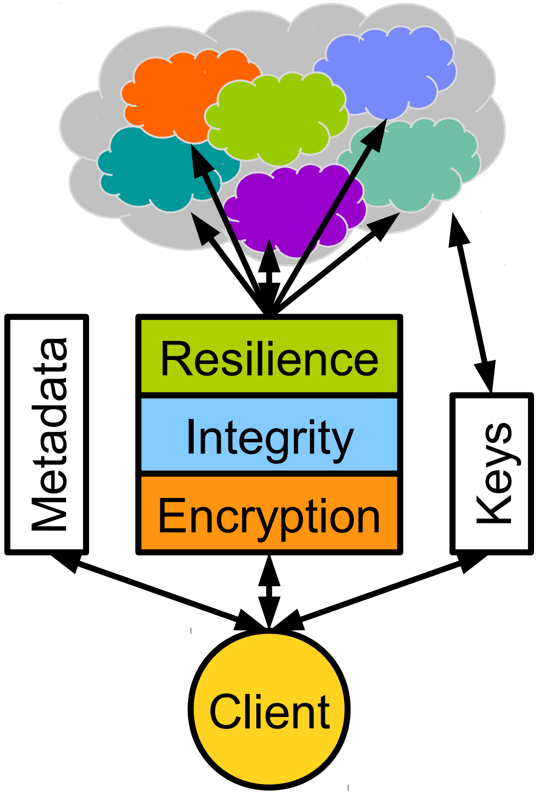
ICStore – Encryption

- Encryption with standard block cipher
- Where do we store keys?



ICStore – Encryption

- Encryption with standard block cipher
- Multiple key management options
  - **Local** keystore
  - Attached to **key server** (OASIS KMIP)
  - In the **intercloud**
    - Uses secret sharing
    - Keys are split and stored across multiple clouds
    - No local keys
    - Qualified set of clouds necessary to recover key
    - Non-qualified set of clouds cannot access data



Reliability

- Most KVS providers do internal replication, so they are already reliable
- Or are they?
  - Gmail temporary mail loss, May 2011
  - Amazon S3 Availability Event: July 20, 2008
  - “Amazon gets 'black eye' from cloud outage” Analysts say downtime hurts Amazon, and cloud computing April 2011

Reliability (cont'd)

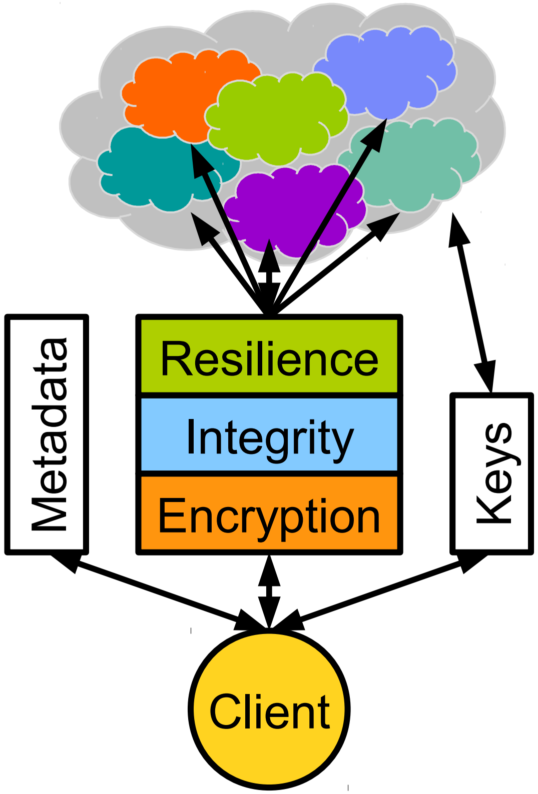
- "Many academics will confess to have made the assumption that failures of components are not correlated. This absolutely unrealistic assumption will come back to haunt you in real life (...)." Werner Vogels, CTO Amazon.com
- Replication is not too effective if
  - Physical failures are correlated (crazy-guy-blows-up-datacentre)
  - What if the same software runs on 100000 nodes (and it has the same bug?)
  - Same security domain (vulnerability)
- Optimal replication across multiple cloud providers

Reliability (cont'd)

- **Objective:**

  use replication across multiple KVS objects (e.g. Amazon S3 AND Microsoft Azure AND Cloudspace) to obtain a more reliable MRMW register

- Read/Write from/to n clouds instead of one
  - Is it that simple?



Reliability (cont'd)

- Let's focus our attention on a single “key”, and see how subsequent replicated put/get operations over 3 KVSes interact with one another
- Working assumptions
  - *2f+1* KVSes, *f* can crash
  - Arbitrary (finite) number of readers/writers
  - Readers/writers can crash
- We will build a solution step-by-step
  - Showing how intermediate solutions are bogus

Reliability (cont'd)

Take 1

- write(key, val)
  - –execute put(key, val) on all KVSes and return when *f+1* have returned
- read(key)
  - –execute read get(key) on all KVSes and return the value returned by *f+1* clouds
- Problems?

Reliability (cont'd)

Take 2

- Use timestamps (sequence numbers); put on multiple keys
- write(key, val)
  - list() and set t0 equal to the highest timestamp seen on a majority of KVSes (or 0 if none)
  - execute put(key.t0+1, val) on all KVSes and return when *f+1* have returned
- read(key)
  - list() and set t0 = the highest timestamp seen on a majority
  - execute read get(key.t0) on all KVSes and return the value returned by the fastest cloud (clouds are honest)
- Problems?

Reliability (cont'd)

Take 3

- Use version = <timestamp, writer IDs>
- write(key, val)
  - –list() and set t0 equal to the highest timestamp seen on a majority (or 0 if none)
  - –execute put(key.t0+1.wid, val) on all KVSes and return when *f+1* have returned
- read(key)
  - –List(), set ver0 = the “highest version” seen on a majority
  - –execute read get(key.ver0) on all KVSes and return the value returned by the fastest cloud (clouds are honest)
- Problems?

Reliability (cont'd)

Take 4

- Garbage collection
  - Who performs it?
  - When?
  - writers may crash but it's ok

Reliability (cont'd)

Take 4

- Garbage collection
- write(key, val)
  - list() and set ver0 = to highest version seen on a majority
  - GC all versions that are there and are < ver0
  - Write (as before)
  - GC ver0
- read(key)
  - As before
- Problems?

Reliability (cont'd)

Take 5

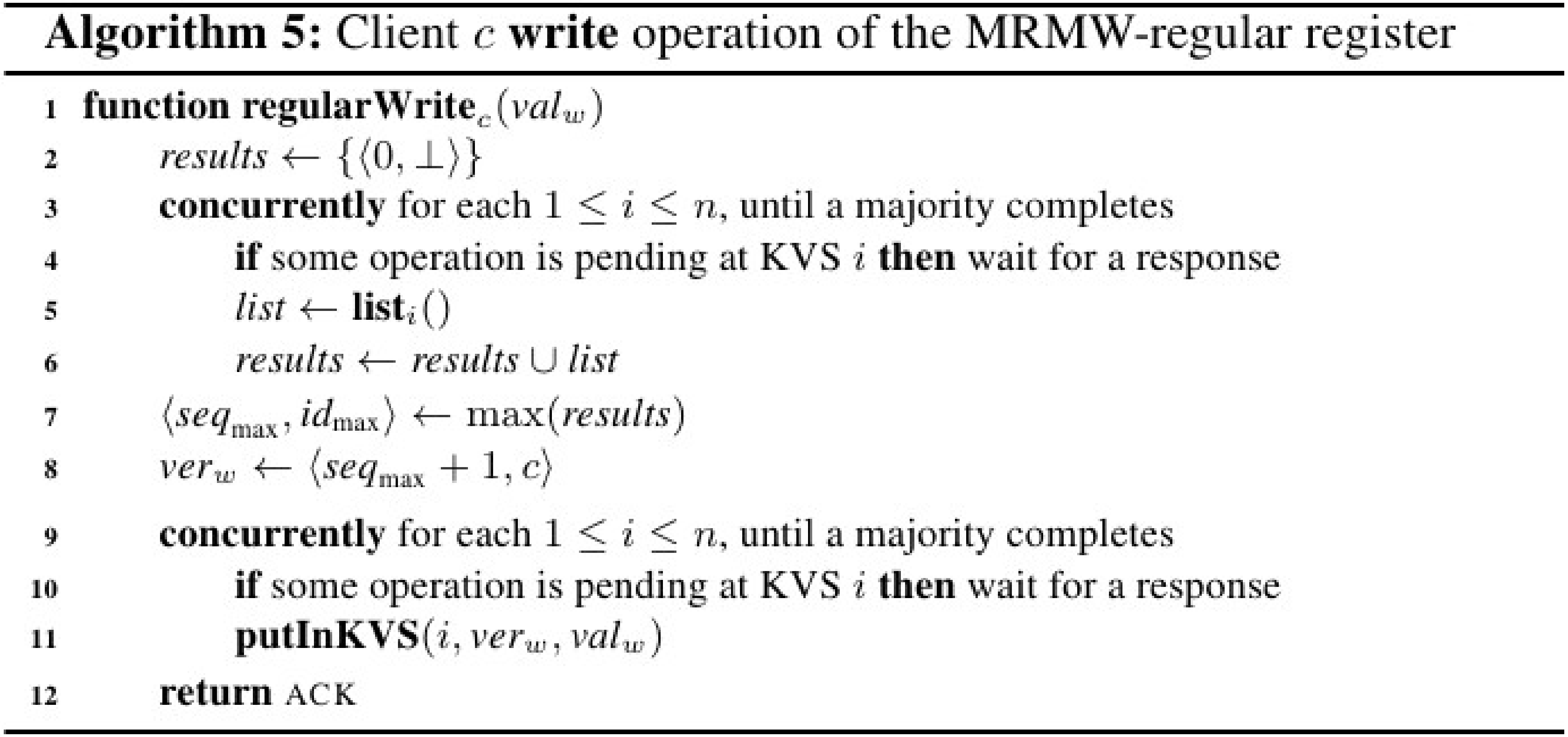
- Reader signaling works
  - “block” GC while reads are in progress
  - Readers need to write, not optimal
- Other ideas?

Reliability (cont'd)

Take 6

- Write twice
  - –Once under “temporary” key (with version number)
  - –Once under “eternal” key (**without** version number)

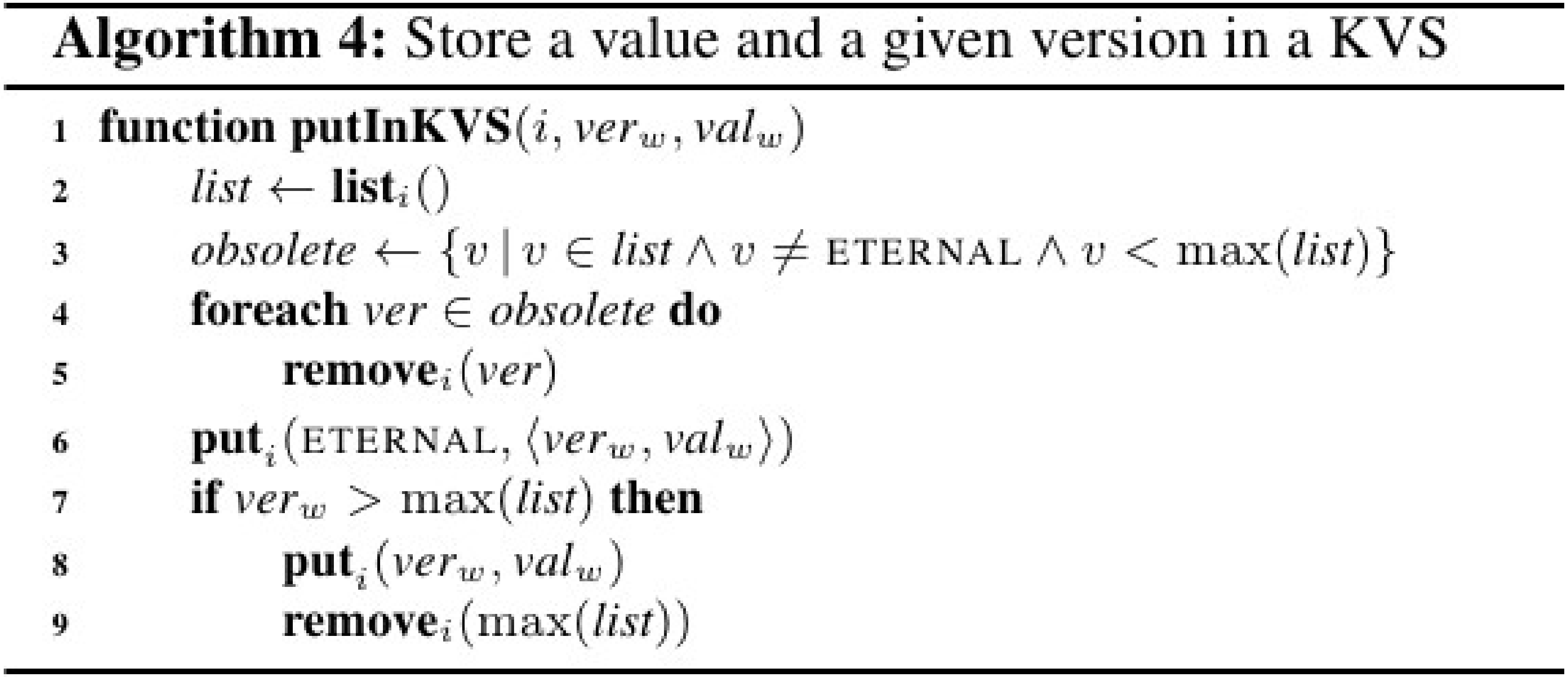
Reliability (cont'd)



Algorithm 5: Client c **write** operation of the MRMW-regular register

```
function regularWrite_c($val_w$)
    $results \leftarrow \{\langle 0, \bot \rangle\}$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if some operation is pending at KVS $i$ then wait for a response
        $list \leftarrow \mathbf{list}_i()$
        $results \leftarrow results \cup list$
    $\langle seq_{\max}, id_{\max} \rangle \leftarrow \max(results)$
    $ver_w \leftarrow \langle seq_{\max} + 1, c \rangle$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if some operation is pending at KVS $i$ then wait for a response
        putInKVS($i, ver_w, val_w$)
    return ACK
```

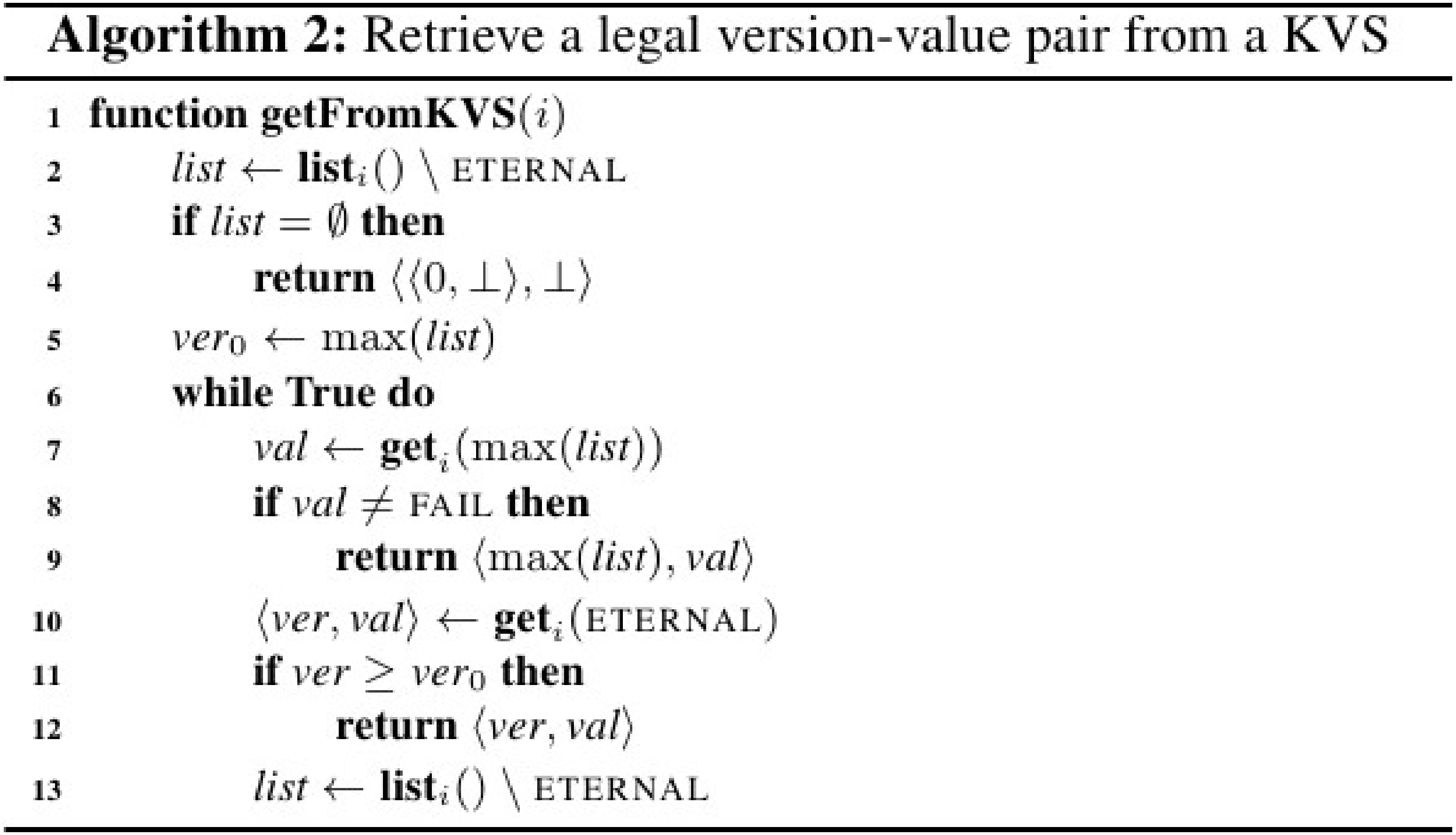
Reliability (cont'd)



Algorithm 4: Store a value and a given version in a KVS

```
function putInKVS($(i, ver_w, val_w)$)
    $list \leftarrow \mathbf{list}_i()$
    $obsolete \leftarrow \{v \mid v \in list \wedge v \neq \text{ETERNAL} \wedge v < \max(list)\}$
    foreach $ver \in obsolete$ do
        $\mathbf{remove}_i(ver)$
    $\mathbf{put}_i(\text{ETERNAL}, \langle ver_w, val_w \rangle)$
    if $ver_w > \max(list)$ then
        $\mathbf{put}_i(ver_w, val_w)$
        $\mathbf{remove}_i(\max(list))$
```

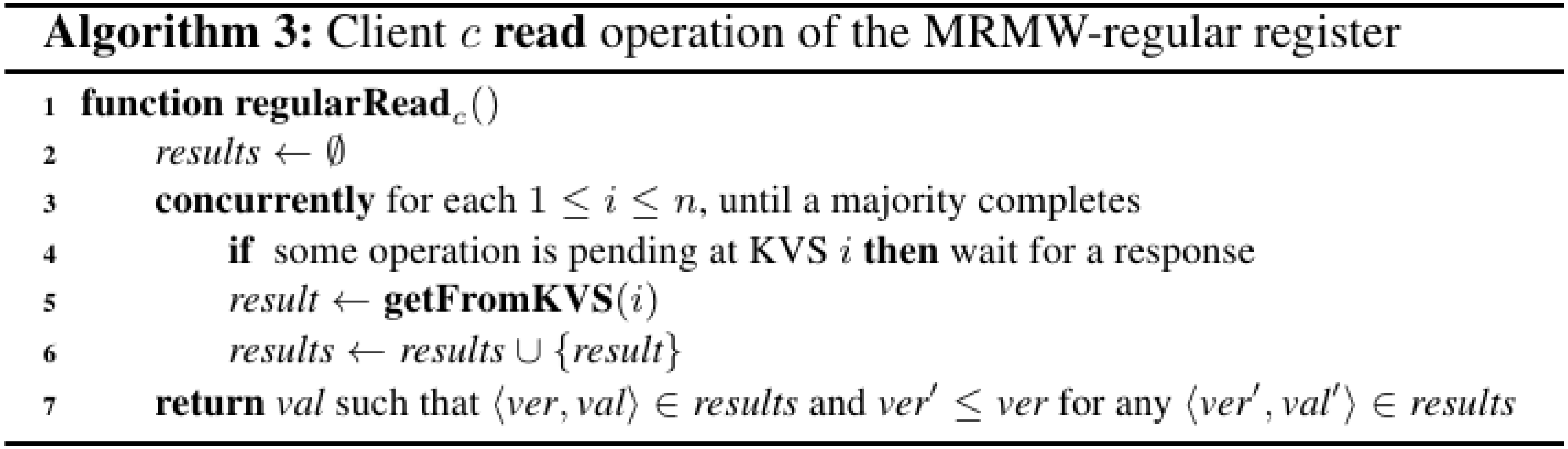
Reliability (cont'd)



Algorithm 2: Retrieve a legal version-value pair from a KVS

```
function getFromKVS($i$)
    $list \leftarrow \mathbf{list}_i() \setminus \textsc{eternal}$
    if $list = \emptyset$ then
        return $\langle\langle 0, \bot\rangle, \bot\rangle$
    $ver_0 \leftarrow \max(list)$
    while True do
        $val \leftarrow \mathbf{get}_i(\max(list))$
        if $val \neq \textsc{fail}$ then
            return $\langle \max(list), val\rangle$
        $\langle ver, val\rangle \leftarrow \mathbf{get}_i(\textsc{eternal})$
        if $ver \geq ver_0$ then
            return $\langle ver, val\rangle$
        $list \leftarrow \mathbf{list}_i() \setminus \textsc{eternal}$
```

Reliability (cont'd)



Algorithm 3: Client c **read** operation of the MRMW-regular register

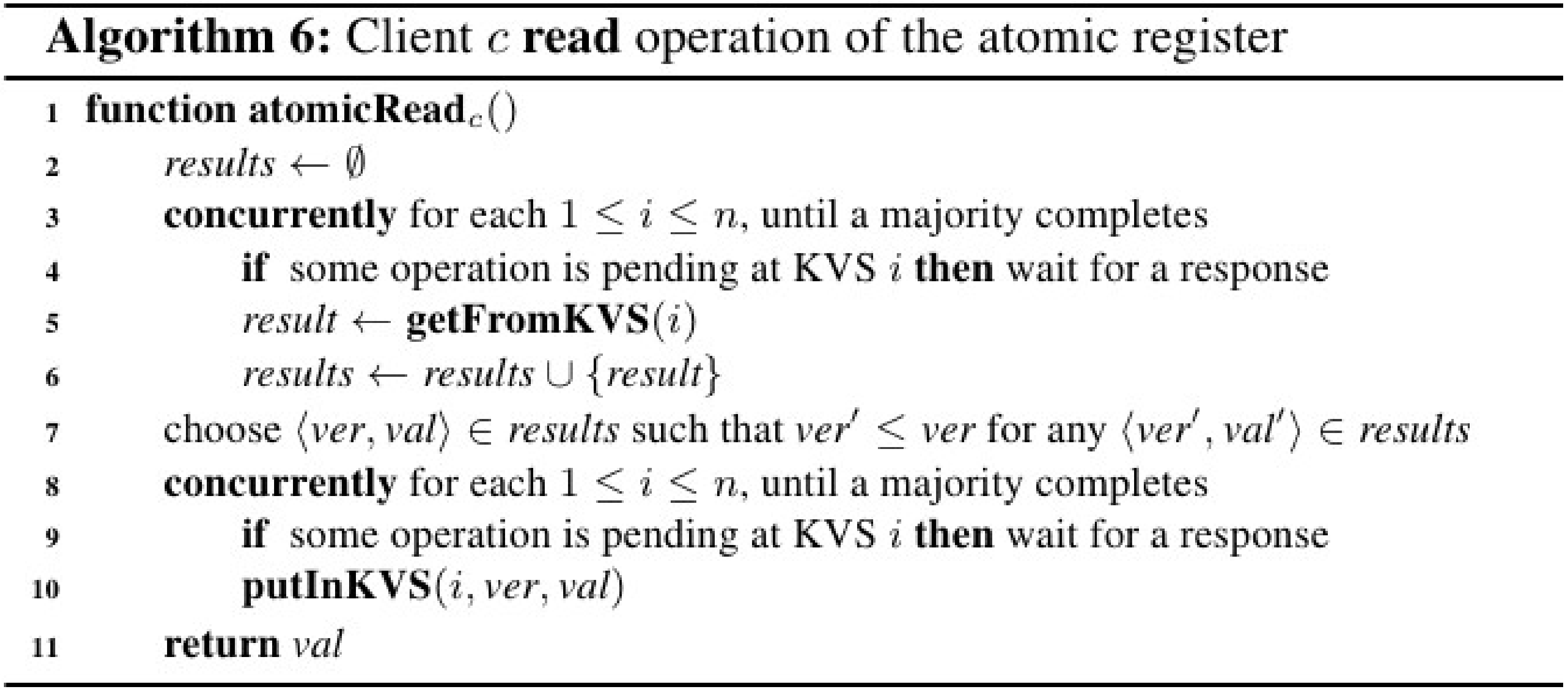
```
function regularRead_c()
    $results \leftarrow \emptyset$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if some operation is pending at KVS $i$ then wait for a response
        $result \leftarrow$ getFromKVS($i$)
        $results \leftarrow results \cup \{result\}$
    return $val$ such that $\langle ver, val \rangle \in results$ and $ver' \leq ver$ for any $\langle ver', val' \rangle \in results$
```

Reliability (cont'd)

- Some observations
  - Regular semantics
  - Wait-freedom
  - Cannot write the temporary before the eternal

Reliability (cont'd)

- Achieving atomicity



Algorithm 6: Client c **read** operation of the atomic register

```
function atomicRead_c()
    $results \leftarrow \emptyset$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if some operation is pending at KVS $i$ then wait for a response
        $result \leftarrow$ getFromKVS($i$)
        $results \leftarrow results \cup \{result\}$
    choose $\langle ver, val \rangle \in results$ such that $ver' \leq ver$ for any $\langle ver', val' \rangle \in results$
    concurrently for each $1 \leq i \leq n$, until a majority completes
        if some operation is pending at KVS $i$ then wait for a response
        putInKVS($i, ver, val$)
    return $val$
```