GoshawkDB

Making Time with Vector Clocks

Matthew Sackman
matthew@goshawkdb.io

https://goshawkdb.io/

1. Have you played with a NoSQL or NewSQL store?

1. Have you played with a NoSQL or NewSQL store?
2. Have you deployed a NoSQL or NewSQL store?

1. Have you played with a NoSQL or NewSQL store?
2. Have you deployed a NoSQL or NewSQL store?
3. Have you studied and know their semantics?

https://aphyr.com/tags/Jepsen

Aphyr Blog Photography Code About

Jepsen: RethinkDB 2.2.3 reconfiguration

In the previous Jepsen analysis of RethinkDB, we tested single-document reads, writes, and conditional writes, under network partitions and process pauses. RethinkDB did not exhibit any nonlinearizable histories in those tests. However, testing with more aggressive failure modes, on both 2.1.5 and 2.2.3, has uncovered a subtle error in Rethink's cluster membership system

Jepsen: RethinkDB 2.1.5

In this Jepsen report, we'll verify RethinkDB's support for linearizable operations using `majority` reads and writes, and explore assorted read and write anomalies when consistency levels are relaxed. This work was funded by RethinkDB, and conducted in accordance with the Jepsen ethics policy.

RethinkDB is an open-source, horizontally scalable document store. Similar to MongoDB,

Jepsen: Percona XtraDB Cluster

Percona's CTO Vadim Tkachenko wrote a response to my Galera Snapshot Isolation post last week. I think Tkachenko may have misunderstood some of my results, and I'd like to clear those up now. I've ported the MariaDB tests to Percona XtraDB Cluster, and would like to confirm that using exclusive write locks on all reads, as Tkachenko recommends, can recover

Jepsen: MariaDB Galera Cluster

Previously, on Jepsen, we saw Chronos fail to run jobs after a network partition. In this post, we'll see MariaDB Galera Cluster allow transactions to read partially committed state.

Galera Cluster extends MySQL (and MySQL's fork, MariaDB) to clusters of machines, all of which support reads and writes. It uses a group

Jepsen: Chronos

Chronos is a distributed task scheduler (cf. cron) for the Mesos cluster management system. In this edition of Jepsen, we'll see how simple network interruptions can permanently disrupt a Chronos+Mesos cluster

Chronos relies on Mesos, which has two flavors of node: master nodes, and slave nodes. Ordinarily in Jepsen we'd refer to these as "primary" and "secondary" or "leader" and

Jepsen: Aerospike

Previously, on Jepsen, we reviewed Elasticsearch's progress in addressing data-loss bugs during network partitions. Today, we'll see Aerospike 3.5.4, an "ACID database", react violently to a basic partition.

Aerospike is a high-performance, distributed, schema-less, KV store, often deployed in caching, analytics, or ad tech environments. Its five-dimensional data model is similar to Bigtable

Jepsen: Elasticsearch 1.5.0

Previously, on Jepsen, we demonstrated stale and dirty reads in MongoDB. In this post, we return to Elasticsearch, which loses data when the network fails, nodes pause, or processes crash.

Nine months ago, in June 2014, we saw Elasticsearch lose both updates and inserted documents during transitive, nontransitive, and even single-node network partitions. Since then,

Jepsen: MongoDB stale reads

In May of 2013, we showed that MongoDB 2.4.3 would lose acknowledged writes at all consistency levels. Every write concern less than MAJORITY loses data by design due to rollbacks–but even WriteConcern.MAJORITY lost acknowledged writes, because when the server encountered a network error, it returned a successful, not a failed, response to the client. Happily, that bug was fixed a few releases later.

Jepsen: Elasticsearch

This post covers Elasticsearch 1.1.0. In the months since its publication, Elasticsearch has added a comprehensive overview of correctness issues and their progress towards fixing some of these bugs.

Previously on Jepsen, we saw RabbitMQ throw

Jepsen: etcd and Consul

In the previous post, we discovered the potential for data loss in RabbitMQ clusters. In this oft-requested installation of the Jepsen series, we'll look at etcd: a new contender in the CP coordination service arena. We'll also discuss Consul's findings with Jepsen.

Like Zookeeper, etcd is designed to store small amounts of strongly-consistent state for coordination between services. It exposes a tree

Jepsen: RabbitMQ

RabbitMQ is a distributed message queue, and is probably the most popular open-source implementation of the AMQP messaging protocol. It supports a wealth of durability, routing, and fanout

Computational techniques in Knossos

Earlier versions of Jepsen found glaring inconsistencies, but missed subtle ones. In particular, Jepsen was not well equipped to distinguish linearizable systems from sequentially or causally consistent ones. When people asked me to analyze systems which claimed to be linearizable, Jepsen could rule out obvious classes of behavior, like dropping writes, but

Strong consistency models

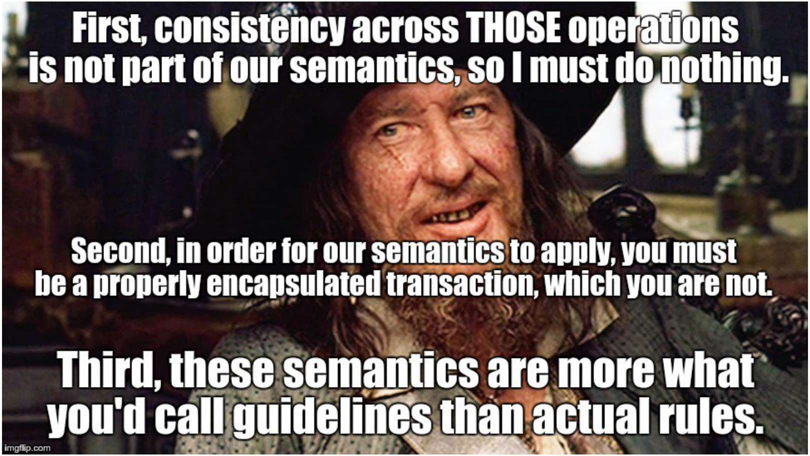
Network partitions are going to happen. Switches, NICs, host hardware, operating systems, disks, virtualization layers, and language runtimes, not

Jepsen: Redis redux

In a recent blog post, antirez detailed a new operation in Redis: `WAIT`. `WAIT` is proposed as an enhancement to Redis' replication protocol to

Jepsen: Strangeloop Hangout

Since the Strangeloop talks won't be available for a few months, I recorded a new version of the talk as a Google Hangout.


First, consistency across THOSE operations
is not part of our semantics, so I must do nothing.
Second, in order for our semantics to apply, you must
be a properly encapsulated transaction, which you are not.
Third, these semantics are more what
you'd call guidelines than actual rules.
imgflip.com

ACID

- Atomic: an operation (transaction) either succeeds or aborts completely - no partial successes
- Consistent: constraints like uniqueness, foreign keys, etc are honoured

- Durable: flushed to disk *before* the client can find out the result

ACID

- Atomic: an operation (transaction) either succeeds or aborts completely - no partial successes
- Consistent: constraints like uniqueness, foreign keys, etc are honoured
- Isolation: the degree to which operations in one transaction can observe actions of concurrent transactions
- Durable: flushed to disk *before* the client can find out the result

Default isolation levels

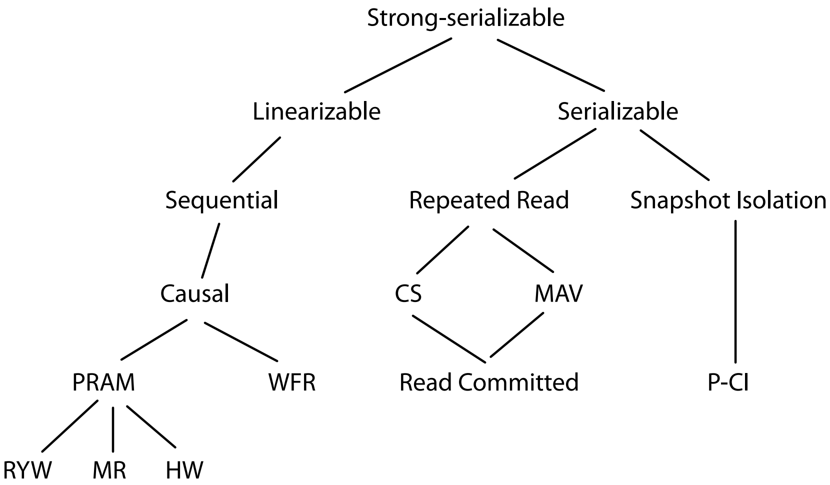
- PostgreSQL:
- Oracle 11g:
- MS SQL Server:
- MySQL InnoDB:

Default isolation levels

- PostgreSQL: Read Committed
- Oracle 11g: Read Committed
- MS SQL Server: Read Committed
- MySQL InnoDB:

Default isolation levels

- PostgreSQL: Read Committed
- Oracle 11g: Read Committed
- MS SQL Server: Read Committed
- MySQL InnoDB: Repeatable Read


Strong-serializable
Linearizable
Serializable
Sequential
Repeated Read
Snapshot Isolation
Causal
CS
MAV
PRAM
WFR
Read Committed
P-CI
RYW
MR
HW

"Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot."

"Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot."

Snapshot isolation is called "serializable" mode in Oracle.

Snapshot Isolation

trouble up mill

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:
  `t1` then `t2`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

```
x, y := 0,0

func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:
  `t1` then `t2`: `x:0, y:1`
  `t2` then `t1`

```
x, y := 0,0

func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`: `x:1, y:0`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:
  - `t1` then `t2`: `x:0, y:1`
  - `t2` then `t1`: `x:1, y:0`
- Snapshot Isolation:

```
x, y := 0,0

func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:
  - `t1` then `t2`: `x:0, y:1`
  - `t2` then `t1`: `x:1, y:0`
- Snapshot Isolation:
  - `t1` || `t2`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`: `x:1, y:0`

- Snapshot Isolation:

  `t1` || `t2`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:
  - `t1` then `t2`: `x:0, y:1`
  - `t2` then `t1`: `x:1, y:0`
- Snapshot Isolation:
  - `t1` || `t2`

```
x, y := 0,0

func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`: `x:1, y:0`

- Snapshot Isolation:

  `t1` || `t2`: `x:1, y:1`

```
x, y := 0,0
```

```
func t1() {
  if x == 0 {
    y = 1
  }
}
```

```
func t2() {
  if y == 0 {
    x = 1
  }
}
```

- Serialized:

  `t1` then `t2`: `x:0, y:1`

  `t2` then `t1`: `x:1, y:0`

- Snapshot Isolation: Write Skew

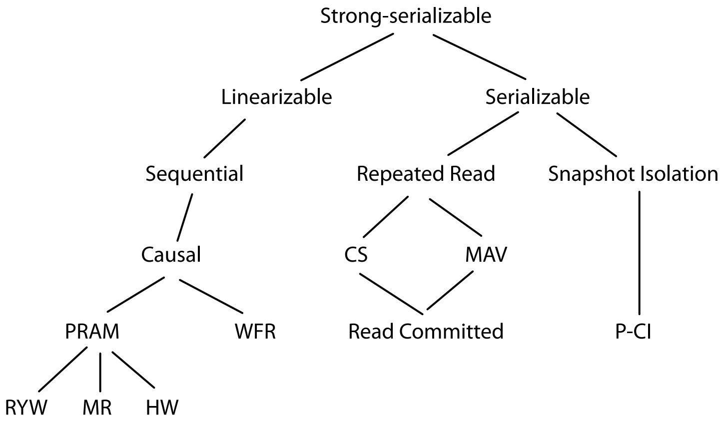
  `t1` || `t2`: `x:1, y:1`

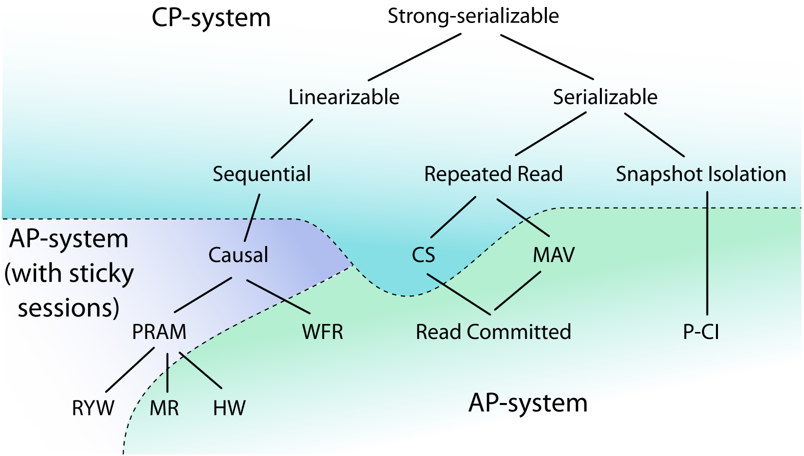
Desired Features

- General purpose transactions

- General purpose transactions
- Strong serializability

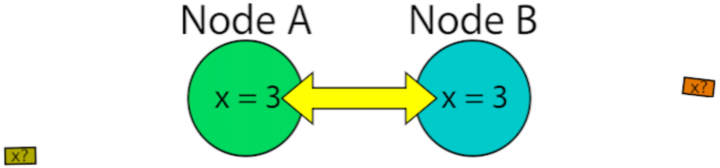
- General purpose transactions
- Strong serializability
- Distribution
- Automatic sharding
- Horizontal scalability
- ...


Strong-serializable
Linearizable
Serializable
Sequential
Repeated Read
Snapshot Isolation
Causal
CS
MAV
PRAM
WFR
Read Committed
P-CI
RYW
MR
HW


CP-system
Strong-serializable
Linearizable
Serializable
Sequential
Repeated Read
Snapshot Isolation
AP-system
(with sticky
sessions)
Causal
CS
MAV
PRAM
WFR
Read Committed
P-CI
RYW
MR
HW
AP-system

$$\text{Possibility of Partitions} \implies \neg(\text{Consistency} \land \text{Availability})$$

Possibility of Partitions $\implies \neg(\text{Consistency} \wedge \text{Availability})$



CAP

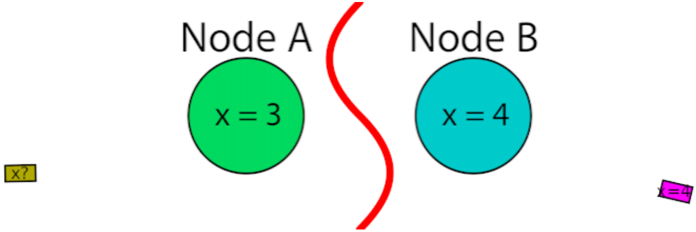
$$\text{Possibility of Partitions} \implies \neg(\text{Consistency} \wedge \text{Availability})$$

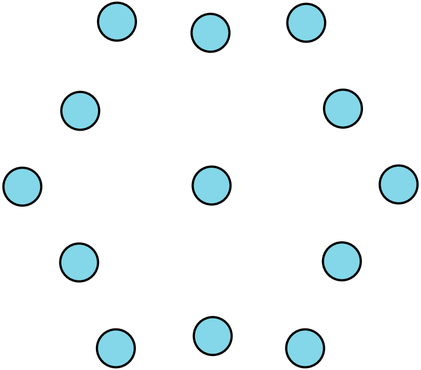
CAP

$$\text{Possibility of Partitions} \implies \neg(\text{Consistency} \wedge \text{Availability})$$

CAP

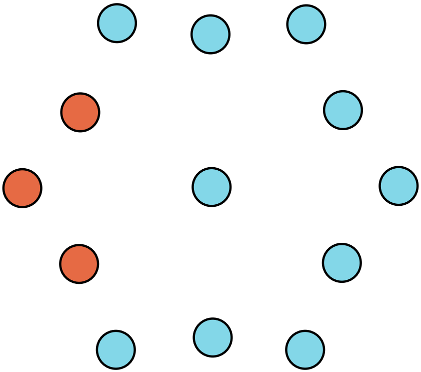
$$\text{Possibility of Partitions} \implies \neg(\text{Consistency} \wedge \text{Availability})$$

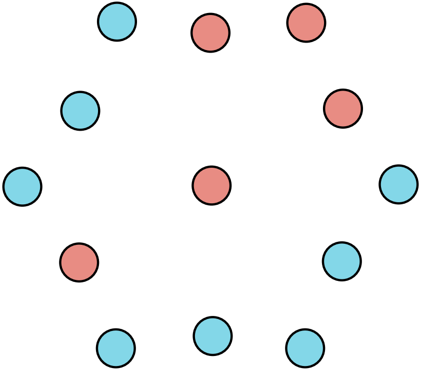


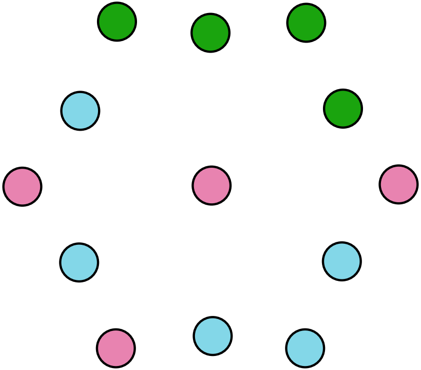


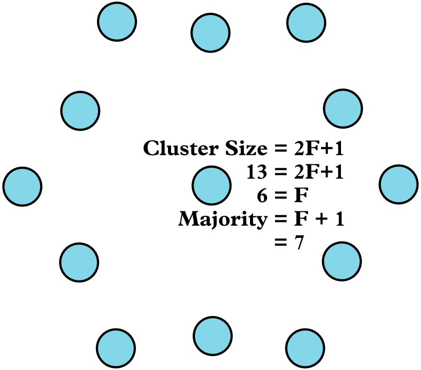
Achieving Consistency

Colours Indicate Connected Nodes





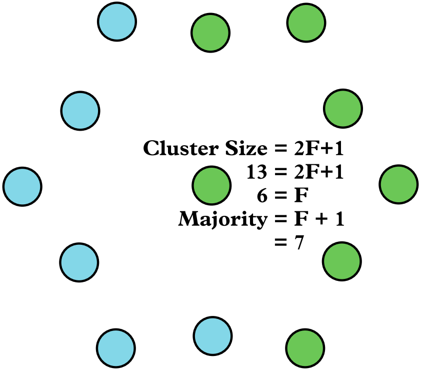




Cluster Size = 2F+1
13 = 2F+1
6 = F
Majority = F + 1
= 7

Achieving Consistency

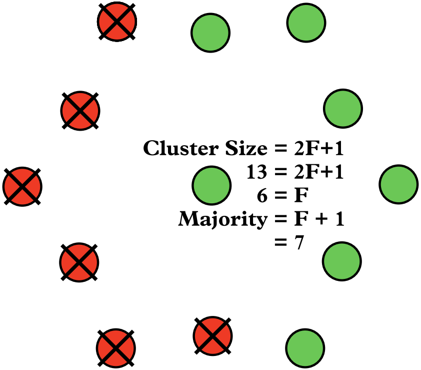
Colours Indicate Connected Nodes



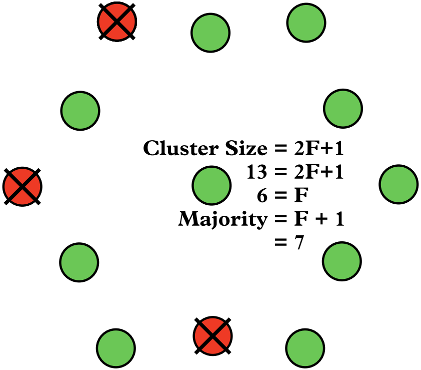
Cluster Size = 2F+1
13 = 2F+1
6 = F
Majority = F + 1
= 7

Achieving Consistency

Colours Indicate Connected Nodes



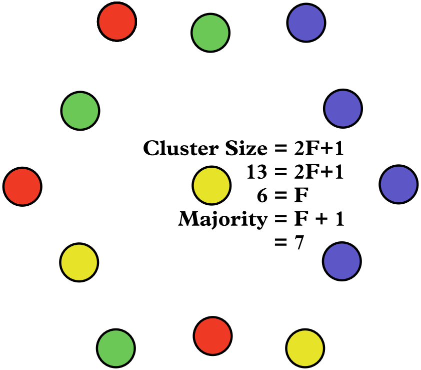
Cluster Size = 2F+1
13 = 2F+1
6 = F
Majority = F + 1
= 7



Cluster Size = 2F+1
13 = 2F+1
6 = F
Majority = F + 1
= 7

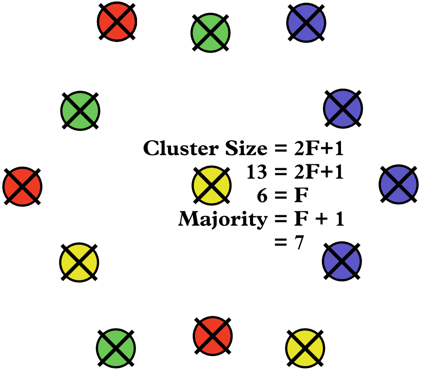
Achieving Consistency

Colours Indicate Connected Nodes



Achieving Consistency

Colours Indicate Connected Nodes



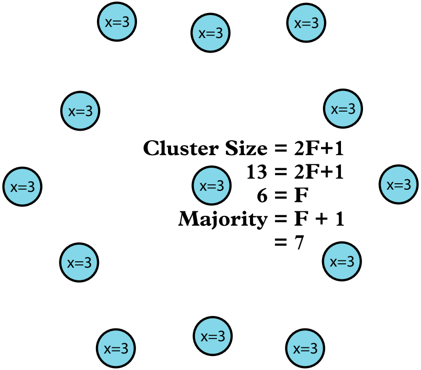
- Strong serializability requires Consistency, so must sacrifice Availability

- Strong serializability requires Consistency, so must sacrifice Availability
- To achieve Consistency, only accept operations if connected to majority

- Strong serializability requires Consistency, so must sacrifice Availability
- To achieve Consistency, only accept operations if connected to majority
- If cluster size is $2F + 1$ then we can withstand no more than F failures

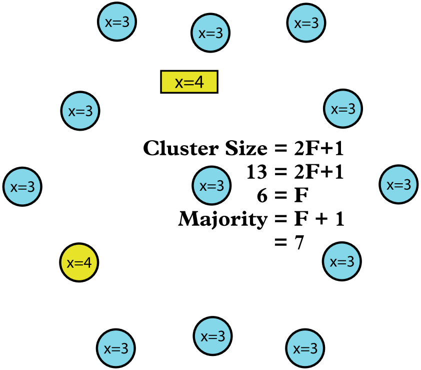
Modifying Values

Colours Indicate Extent of Txn Write



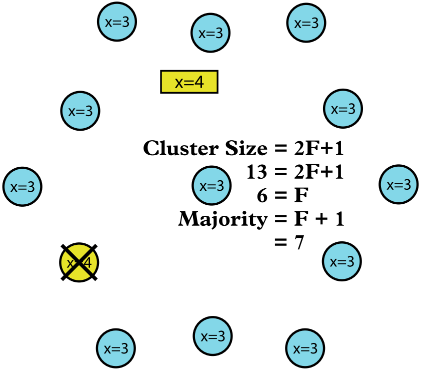
Modifying Values

Colours Indicate Extent of Txn Write



Modifying Values

Colours Indicate Extent of Txn Write



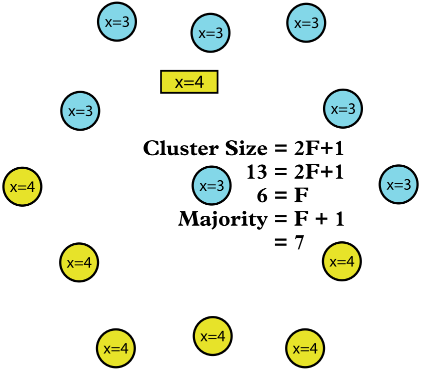
Cluster Size = 2F+1

13 = 2F+1

6 = F

Majority = F + 1

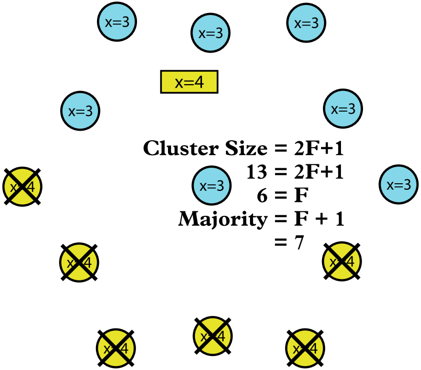
= 7



$$
\begin{aligned}
\textbf{Cluster Size} &= \mathbf{2F+1} \\
\mathbf{13} &= \mathbf{2F+1} \\
\mathbf{6} &= \mathbf{F} \\
\textbf{Majority} &= \mathbf{F+1} \\
&= 7
\end{aligned}
$$

Modifying Values

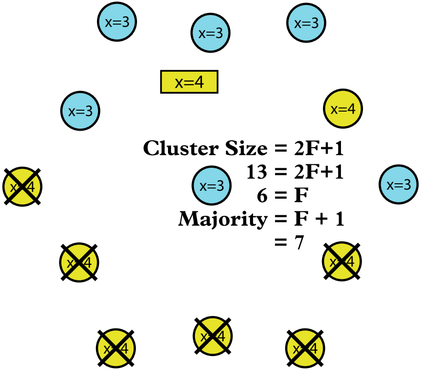
Colours Indicate Extent of Txn Write



$$
\begin{aligned}
\textbf{Cluster Size} &= \mathbf{2F+1}\\
\mathbf{13} &= \mathbf{2F+1}\\
\mathbf{6} &= \mathbf{F}\\
\textbf{Majority} &= \mathbf{F + 1}\\
&= 7
\end{aligned}
$$

Modifying Values

Colours Indicate Extent of Txn Write

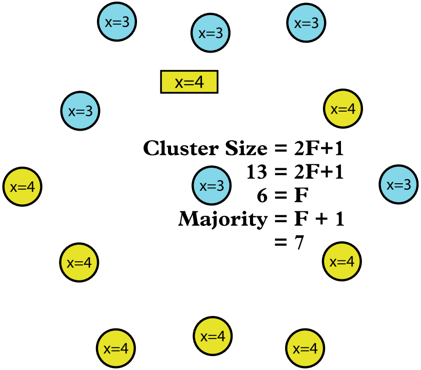


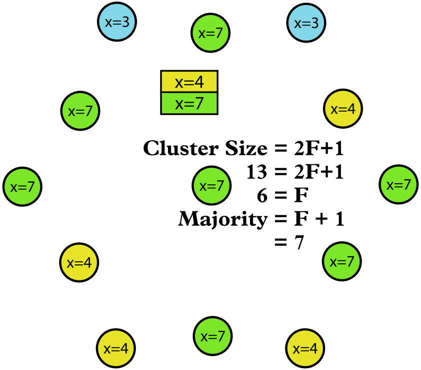
Learnings 2

- Strong serializability requires Consistency, so must sacrifice Availability
- To achieve Consistency, only accept operations if connected to majority
- If cluster size is $2F + 1$ then we can withstand no more than F failures
- Writes must go to $F + 1$ nodes

Reading Values

Colours Indicate Extent of Txn Write

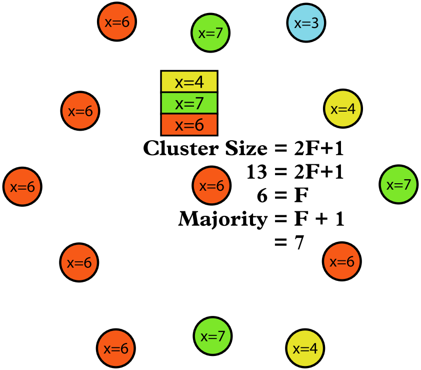


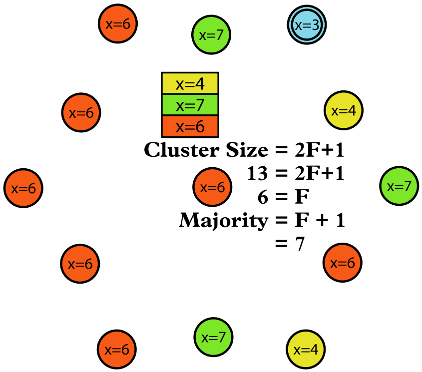


Cluster Size = 2F+1
13 = 2F+1
6 = F
Majority = F + 1
= 7

Reading Values

Colours Indicate Extent of Txn Write

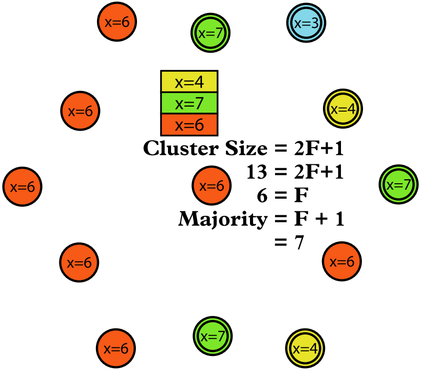




$$
\begin{aligned}
\textbf{Cluster Size} &= 2F+1\\
13 &= 2F+1\\
6 &= F\\
\textbf{Majority} &= F+1\\
&= 7
\end{aligned}
$$

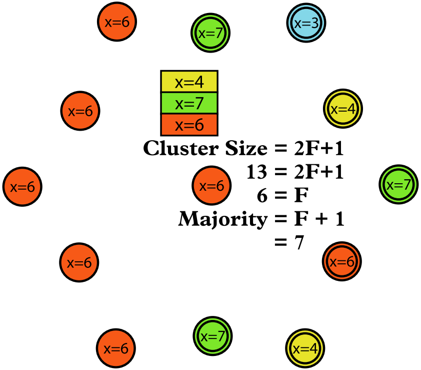
Reading Values

Colours Indicate Extent of Txn Write



Reading Values

Colours Indicate Extent of Txn Write



Learnings 3

- Strong serializability requires Consistency, so must sacrifice Availability
- To achieve Consistency, only accept operations if connected to majority
- If cluster size is $2F + 1$ then we can withstand no more than F failures
- Writes must go to $F + 1$ nodes
- Reads must read from $F + 1$ nodes and be able to order results

1. Client submits txn

Txn Processing in Distributed Databases

1. Client submits txn
2. Node(s) vote on txn

1. Client submits txn
2. Node(s) vote on txn
3. Node(s) reach consensus on txn outcome

Txn Processing in Distributed Databases

1. Client submits txn
2. Node(s) vote on txn
3. Node(s) reach consensus on txn outcome
4. Client is informed of outcome

Txn Processing in Distributed Databases

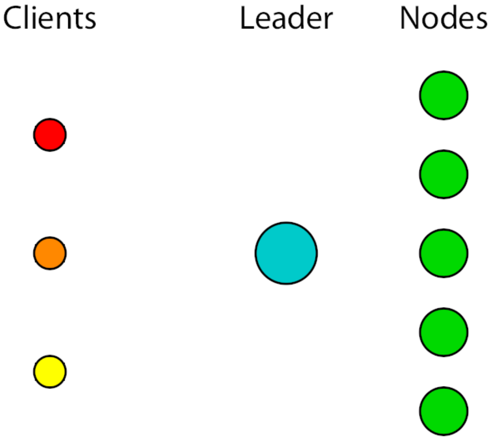
1. Client submits txn
2. Node(s) vote on txn
3. Node(s) reach consensus on txn outcome
4. Client is informed of outcome

Most important thing is all nodes agree on the order of transactions

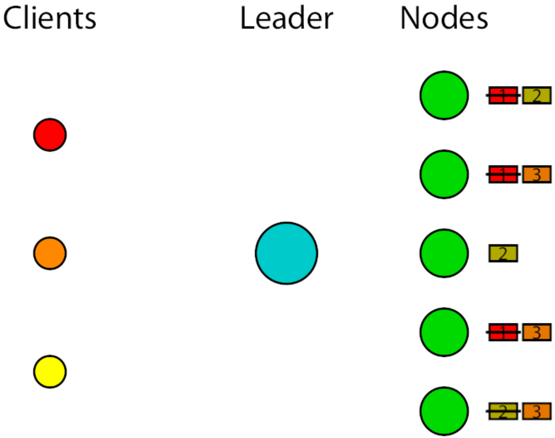
Txn Processing in Distributed Databases

1. Client submits txn
2. Node(s) vote on txn
3. Node(s) reach consensus on txn outcome
4. Client is informed of outcome

Most important thing is all nodes agree on the order of transactions
(focus for the rest of this talk!)


Clients
Leader
Nodes

Leader Based Ordering



- Only leader votes on whether txn commits or aborts

- Only leader votes on whether txn commits or aborts
- Therefore leader must know everything

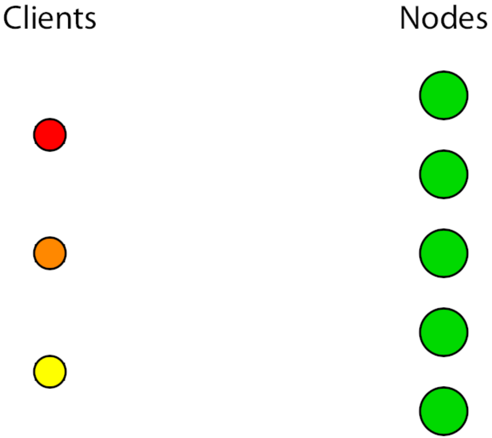
- Only leader votes on whether txn commits or aborts
- Therefore leader must know everything
- If leader fails, a new leader will be elected from remaining nodes

Leader Based Ordering

- Only leader votes on whether txn commits or aborts
- Therefore leader must know everything
- If leader fails, a new leader will be elected from remaining nodes
- Therefore all nodes must know everything

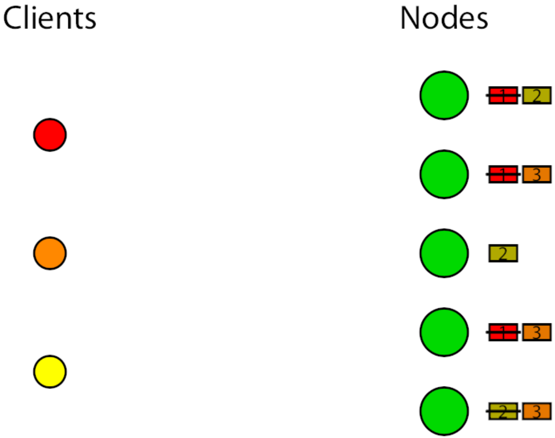
- Only leader votes on whether txn commits or aborts
- Therefore leader must know everything
- If leader fails, a new leader will be elected from remaining nodes
- Therefore all nodes must know everything
- Fine for small clusters, but scaling issues when clusters get big

Client Clock Based Ordering



Client Clock Based Ordering

Client Clock Based Ordering



- Nodes receive txns and must vote on txn outcome and then consensus must be reached (not shown)

- Nodes receive txns and must vote on txn outcome and then consensus must be reached (not shown)
- Clients are responsible for applying an increasing clock value to txns

Client Clock Based Ordering

- Nodes receive txns and must vote on txn outcome and then consensus must be reached (not shown)
- Clients are responsible for applying an increasing clock value to txns
- If a client's clock races then it can prevent other clients from getting txns submitted

Client Clock Based Ordering

- Nodes receive txns and must vote on txn outcome and then consensus must be reached (not shown)
- Clients are responsible for applying an increasing clock value to txns
- If a client's clock races then it can prevent other clients from getting txns submitted
- So must be very careful to try and keep clocks running at the same rate

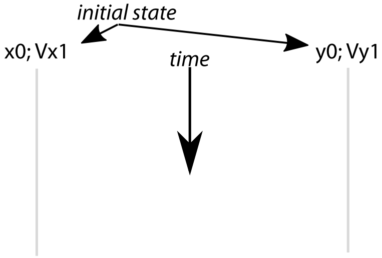
Client Clock Based Ordering

- Nodes receive txns and must vote on txn outcome and then consensus must be reached (not shown)
- Clients are responsible for applying an increasing clock value to txns
- If a client's clock races then it can prevent other clients from getting txns submitted
- So must be very careful to try and keep clocks running at the same rate
- No possibility to reorder transactions at all to maximise commits

?m	receive message m (sender unspecified)
!m	send message m (destination unspecified)
t3	transaction with id 3
r[x1]	reads x at version 1
w[y]	writes some value to y
Vx2y1	vector clock with x=2, y=1

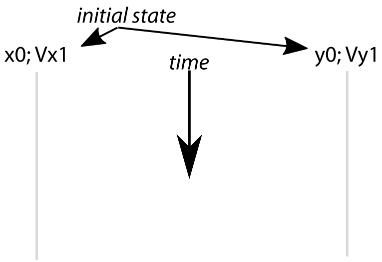
$$\begin{aligned} V_1 < V_2 \triangleq & \forall x \in \mathrm{dom}(V_1 \cup V_2) : V_1[x] \leq V_2[x] \\ & \wedge \exists y \in \mathrm{dom}(V_1 \cup V_2) : V_1[y] < V_2[y] \end{aligned}$$

Simple Transaction



?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

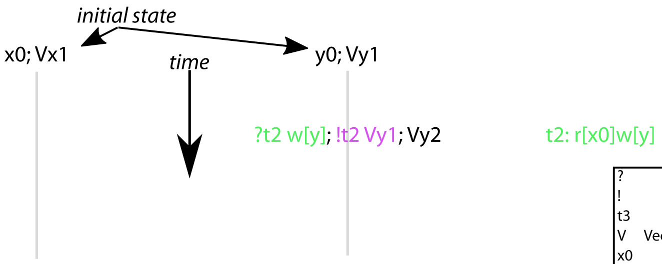
Simple Transaction



t2: r[x0]w[y]

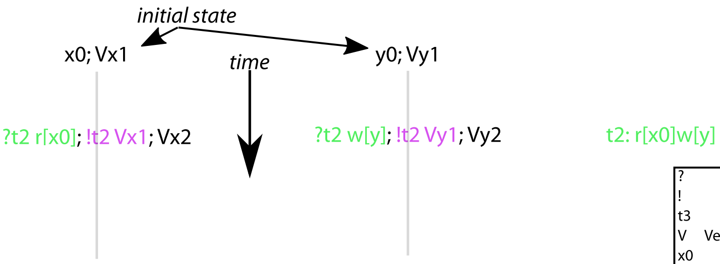
?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Simple Transaction

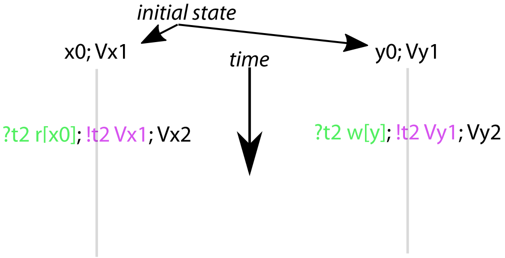


?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Simple Transaction

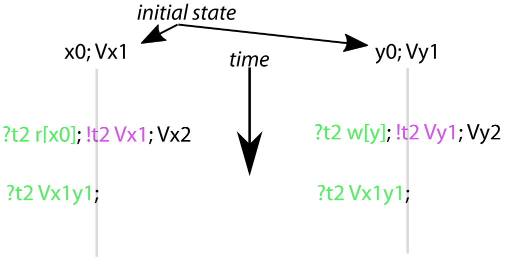


?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x



t2: r[x0]w[y]   Vx1y1

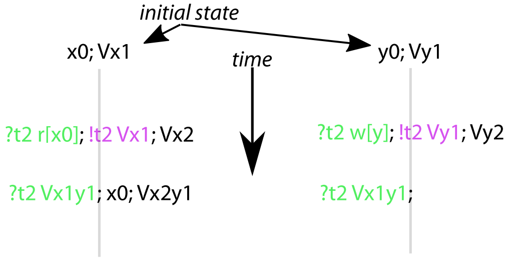
?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x



t2: r[x0]w[y] Vx1y1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

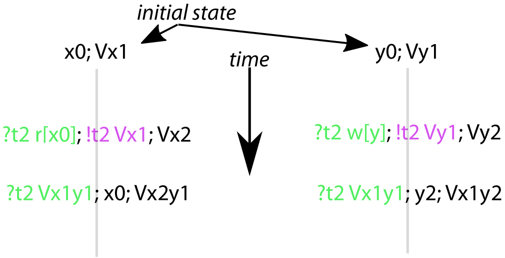
Simple Transaction



t2: r[x0]w[y]   Vx1y1

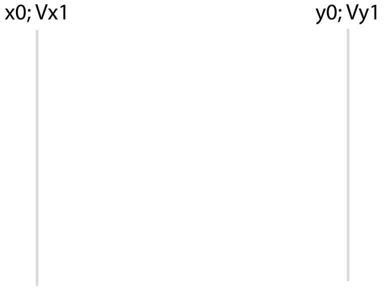
?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Simple Transaction



t2: r[x0]w[y]   Vx1y1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x



t1 w[x]w[y]

t2 w[x]w[y]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Two Writes

x0; Vx1

?t1 w[x]; !t1 Vx1; Vx2

y0; Vy1

?t2 w[y]; !t2 Vy1; Vy2

t1 w[x]w[y]

t2 w[x]w[y]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Two Writes

x0; Vx1

?t1 w[x]; !t1 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3

y0; Vy1

?t2 w[y]; !t2 Vy1; Vy2
?t1 w[y]; !t1 Vy2; Vy3

t1 w[x]w[y]
t2 w[x]w[y]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Two Writes

x0; Vx1

?t1 w[x]; !t1 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3

y0; Vy1

?t2 w[y]; !t2 Vy1; Vy2
?t1 w[y]; !t1 Vy2; Vy3

t1 w[x]w[y] Vx1y2
t2 w[x]w[y] Vx2y1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Two Writes

x0; Vx1

?t1 w[x]; !t1 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3

?t1 Vx1y2; x1; Vx3y2

y0; Vy1

?t2 w[y]; !t2 Vy1; Vy2
?t1 w[y]; !t1 Vy2; Vy3

?t1 Vx1y2; y1; Vx1y3

t1 w[x]w[y] Vx1y2
t2 w[x]w[y] Vx2y1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

x0; Vx1

?t1 w[x]; !t1 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3

?t1 Vx1y2; x1; Vx3y2
?t2 Vx2y1; x?; Vx3y2

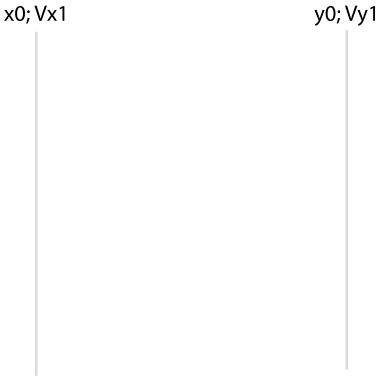
y0; Vy1

?t2 w[y]; !t2 Vy1; Vy2
?t1 w[y]; !t1 Vy2; Vy3

?t1 Vx1y2; y1; Vx1y3
?t2 Vx2y1; y?; Vx2y3

t1 w[x]w[y] Vx1y2
t2 w[x]w[y] Vx2y1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x



t1 w[x]w[y]

t2 w[x]

t3 w[x]w[y]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

x0; Vx1

y0; Vy1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

t1 w[x]w[y]
t2 w[x]
t3 w[x]w[y]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Three Writes

x0; Vx1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

y0; Vy1

?t1 w[y]; !t1 Vy1; Vy2
?t3 w[y]; !t3 Vy2; Vy3

t1 w[x]w[y]
t2 w[x]
t3 w[x]w[y]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Three Writes

x0; Vx1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

y0; Vy1

?t1 w[y]; !t1 Vy1; Vy2
?t3 w[y]; !t3 Vy2; Vy3

t1 w[x]w[y]	Vx3y1
t2 w[x]	Vx2
t3 w[x]w[y]	Vx1y2

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

x0; Vx1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

y0; Vy1

?t1 w[y]; !t1 Vy1; Vy2
?t3 w[y]; !t3 Vy2; Vy3

?t3 Vx1y2; y3; Vx1y3
?t1 Vx3y1; y3; Vx3y3

t1 w[x]w[y]	Vx3y1
t2 w[x]	Vx2
t3 w[x]w[y]	Vx1y2

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Three Writes

x0; Vx1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

y0; Vy1

?t1 w[y]; !t1 Vy1; Vy2
?t3 w[y]; !t3 Vy2; Vy3

?t3 Vx1y2; y3; Vx1y3
?t1 Vx3y1; y3; Vx3y3

t1 < t3

t1 w[x]w[y]	Vx3y1
t2 w[x]	Vx2
t3 w[x]w[y]	Vx1y2

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

x0; Vx1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

?t3 Vx1y2; x3; Vx4y2
?t2 Vx2; x2; Vx4y2
?t1 Vx3y1; x1; Vx4y2

y0; Vy1

?t1 w[y]; !t1 Vy1; Vy2
?t3 w[y]; !t3 Vy2; Vy3

?t3 Vx1y2; y3; Vx1y3
?t1 Vx3y1; y3; Vx3y3

t1 < t3

t1 w[x]w[y]	Vx3y1
t2 w[x]	Vx2
t3 w[x]w[y]	Vx1y2

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Three Writes

x0; Vx1

?t3 w[x]; !t3 Vx1; Vx2
?t2 w[x]; !t2 Vx2; Vx3
?t1 w[x]; !t1 Vx3; Vx4

?t3 Vx1y2; x3; Vx4y2
?t2 Vx2; x2; Vx4y2
?t1 Vx3y1; x1; Vx4y2

t3 < t2 < t1

y0; Vy1

?t1 w[y]; !t1 Vy1; Vy2
?t3 w[y]; !t3 Vy2; Vy3

?t3 Vx1y2; y3; Vx1y3
?t1 Vx3y1; y3; Vx3y3

t1 < t3

t1 w[x]w[y]	Vx3y1
t2 w[x]	Vx2
t3 w[x]w[y]	Vx1y2

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

The Dumb Approach Doesn't Work

- Changing state when receiving a txn seems to be a very bad idea

The Dumb Approach Doesn't Work

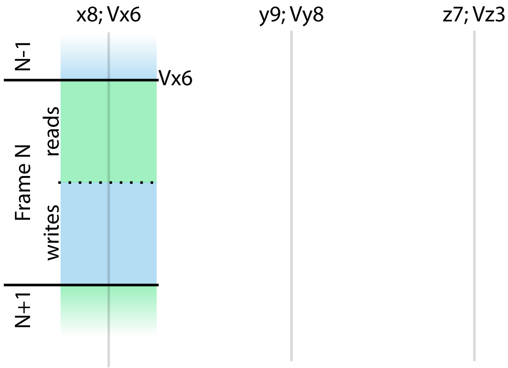
- Changing state when receiving a txn seems to be a very bad idea
- Maybe only change state when receiving the outcome of a vote

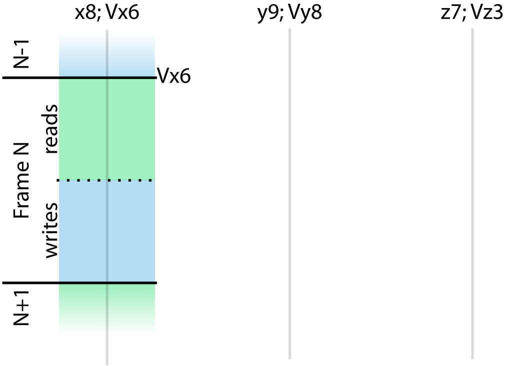
The Dumb Approach Doesn't Work

- Changing state when receiving a txn seems to be a very bad idea
- Maybe only change state when receiving the outcome of a vote
- And don't vote on txns until we know it's safe to do so

- Divide time into frames. First half of frame is reads, second half writes.

- Divide time into frames. First half of frame is reads, second half writes.
- Within a frame, we don't care about order of reads,
- but all reads must come after writes of previous frame,
- all writes must come after reads of this frame,
- all writes must be totally ordered within the frame - must know which write comes last.

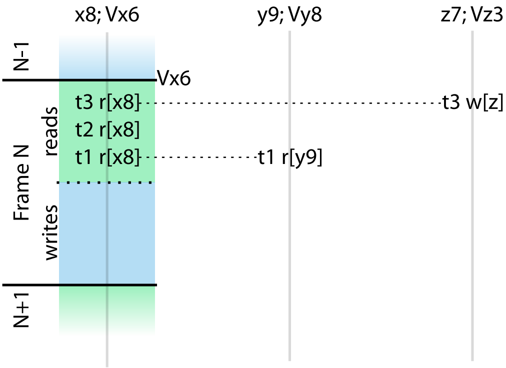

x8; Vx6
y9; Vy8
z7; Vz3
N-1
Vx6
reads
Frame N
writes
N+1



t1 r[x8]r[y9]

t2 r[x8]

t3 r[x8]w[z]

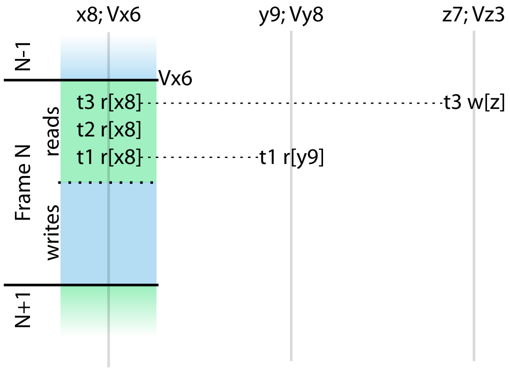


t1 r[x8]r[y9]

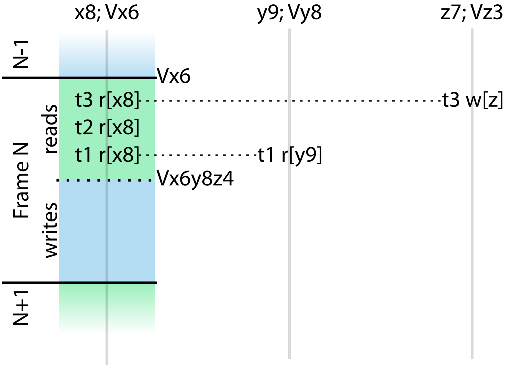
t2 r[x8]

t3 r[x8]w[z]

Frames & Dependencies

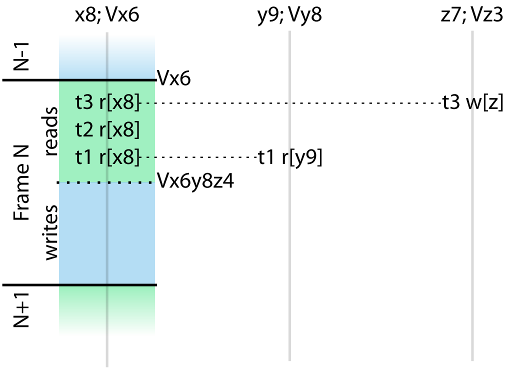


t1 r[x8]r[y9]	Vx6y8
t2 r[x8]	Vx6
t3 r[x8]w[z]	Vx6z3



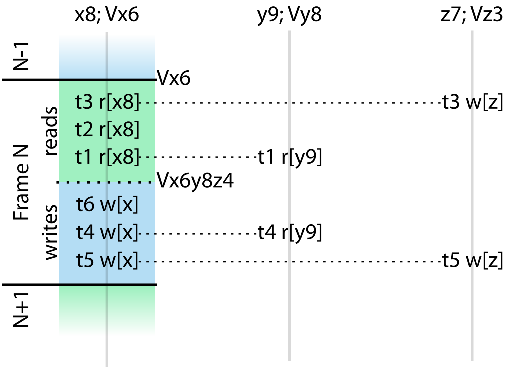
t1 r[x8]r[y9]	Vx6y8
t2 r[x8]	Vx6
t3 r[x8]w[z]	Vx6z3

Frames & Dependencies

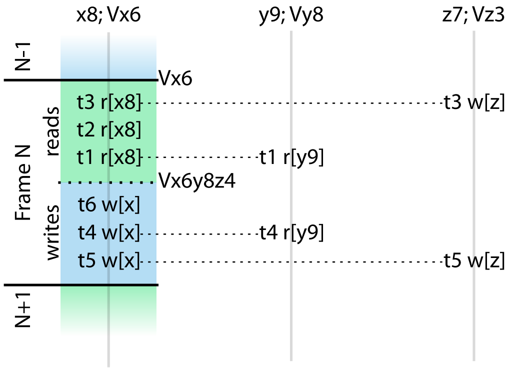


t1 r[x8]r[y9]	Vx6y8
t2 r[x8]	Vx6
t3 r[x8]w[z]	Vx6z3
t4 w[y]r[y9]	
t5 w[x]w[z]	
t6 w[x]	

Frames & Dependencies



t1 r[x8]r[y9]	Vx6y8
t2 r[x8]	Vx6
t3 r[x8]w[z]	Vx6z3
t4 w[y]r[y9]	
t5 w[x]w[z]	
t6 w[x]	



t1 r[x8]r[y9]	Vx6y8
t2 r[x8]	Vx6
t3 r[x8]w[z]	Vx6z3
t4 w[y]r[y9]	Vx6y8z4
t5 w[x]w[z]	Vx6y8z?
t6 w[x]	Vx6y8z4

Calculating the Write Clock from Reads

- Merge all read clocks together
- Add 1 to result for every object that was written by txns in our frame's reads

Calculating the Frame Winner
& Next Frame's Read Clock

- Partition write results by local clock elem, and within that by txn id
- Each clock inherits missing clock elems from above
- Then sort each partition first by clock (now all same length), then by txn id
- Next frame starts with winner's clock, +1 for all writes

Calculating the Frame Winner

& Next Frame's Read Clock

- Partition write results by local clock elem, and within that by txn id
- Each clock inherits missing clock elems from above
- Then sort each partition first by clock (now all same length), then by txn id
- Next frame starts with winner's clock, +1 for all writes
- Guarantees no *concurrent* vector clocks (proof in progress!)

Calculating the Frame Winner
& Next Frame's Read Clock

- Partition write results by local clock elem, and within that by txn id
- Each clock inherits missing clock elems from above
- Then sort each partition first by clock (now all same length), then by txn id
- Next frame starts with winner's clock, +1 for all writes
- Guarantees no *concurrent* vector clocks (proof in progress!)
- Many details elided! (deadlock freedom, etc)

x0; Vx1

?t1 w[x]; !t1 Vx1

?t2 w[x]; !t2 Vx1

y0; Vy1

?t2 w[y]; !t2 Vy1

?t3 w[y]; !t3 Vy1

z0; Vz1

?t3 w[z]; !t3 Vz1

t1: w[x]w[z]

t2: w[x]w[y]

t3: w[y]w[z]

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks

x0; Vx1

?t1 w[x]; !t1 Vx1

?t2 w[x]; !t2 Vx1

y0; Vy1

?t2 w[y]; !t2 Vy1

?t3 w[y]; !t3 Vy1

?t3 Vy1z1

z0; Vz1

?t3 w[z]; !t3 Vz1

?t3 Vy1z1

frame z[3]; Vy2z2

t1: w[x]w[z]

t2: w[x]w[y]

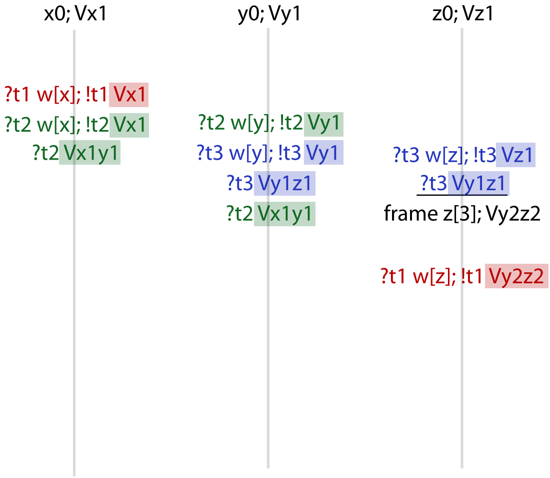
t3: w[y]w[z] Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks

x0; Vx1	y0; Vy1	z0; Vz1	
?t1 w[x]; !t1 Vx1			t1: w[x]w[z]
?t2 w[x]; !t2 Vx1	?t2 w[y]; !t2 Vy1		t2: w[x]w[y] Vx1y1
?t2 Vx1y1	?t3 w[y]; !t3 Vy1	?t3 w[z]; !t3 Vz1	t3: w[y]w[z] Vy1z1
	?t3 Vy1z1	?t3 Vy1z1	
	?t2 Vx1y1	frame z[3]; Vy2z2	

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x



t1: w[x]w[z]

t2: w[x]w[y] Vx1y1

t3: w[y]w[z] Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

x0; Vx1

?t1 w[x]; !t1 Vx1
?t2 w[x]; !t2 Vx1
?t2 Vx1y1
?t1 Vx1y2z2

y0; Vy1

?t2 w[y]; !t2 Vy1
?t3 w[y]; !t3 Vy1
?t3 Vy1z1
?t2 Vx1y1

z0; Vz1

?t3 w[z]; !t3 Vz1
?t3 Vy1z1
frame z[3]; Vy2z2
?t1 w[z]; !t1 Vy2z2
?t1 Vx1y2z2

t1: w[x]w[z] Vx1y2z2
t2: w[x]w[y] Vx1y1
t3: w[y]w[z] Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks

x0; Vx1

?t1 w[x]; !t1 Vx1

?t2 w[x]; !t2 Vx1

?t2 Vx1y1

?t1 Vx1y2z2

y0; Vy1

?t2 w[y]; !t2 Vy1

?t3 w[y]; !t3 Vy1

?t3 Vy1z1

?t2 Vx1y1

2 < 3?

z0; Vz1

?t3 w[z]; !t3 Vz1

?t3 Vy1z1

frame z[3]; Vy2z2

?t1 w[z]; !t1 Vy2z2

?t1 Vx1y2z2

3 < 1

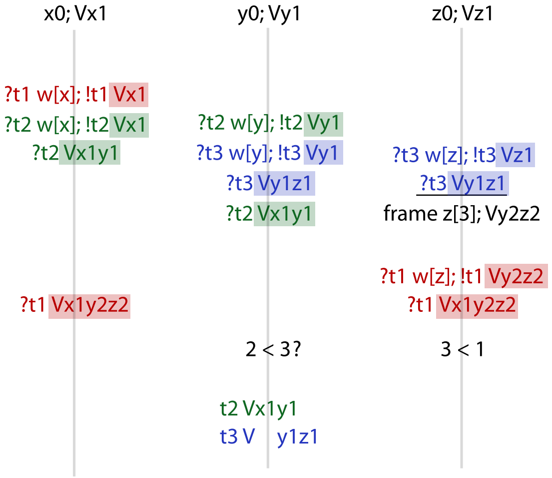
t1: w[x]w[z] Vx1y2z2

t2: w[x]w[y] Vx1y1

t3: w[y]w[z] Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks



t1: w[x]w[z]	Vx1y2z2
t2: w[x]w[y]	Vx1y1
t3: w[y]w[z]	Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks

x0; Vx1

?t1 w[x]; !t1 Vx1

?t2 w[x]; !t2 Vx1

?t2 Vx1y1

?t1 Vx1y2z2

y0; Vy1

?t2 w[y]; !t2 Vy1

?t3 w[y]; !t3 Vy1

?t3 Vy1z1

?t2 Vx1y1

2 < 3

t2 Vx1y1

t3 Vx1y1z1

z0; Vz1

?t3 w[z]; !t3 Vz1

?t3 Vy1z1

frame z[3]; Vy2z2

?t1 w[z]; !t1 Vy2z2

?t1 Vx1y2z2

3 < 1

t1: w[x]w[z] Vx1y2z2

t2: w[x]w[y] Vx1y1

t3: w[y]w[z] Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks

x0; Vx1

?t1 w[x]; !t1 Vx1
?t2 w[x]; !t2 Vx1
?t2 Vx1y1

?t1 Vx1y2z2

2 < 1?

t1 Vx1y2z2
t2 Vx1y1

y0; Vy1

?t2 w[y]; !t2 Vy1
?t3 w[y]; !t3 Vy1
?t3 Vy1z1
?t2 Vx1y1

2 < 3

t2 Vx1y1
t3 Vx1y1z1

z0; Vz1

?t3 w[z]; !t3 Vz1
?t3 Vy1z1

frame z[3]; Vy2z2

?t1 w[z]; !t1 Vy2z2
?t1 Vx1y2z2

3 < 1

t1: w[x]w[z] Vx1y2z2
t2: w[x]w[y] Vx1y1
t3: w[y]w[z] Vy1z1

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

Transitive Vector Clocks

x0; Vx1	y0; Vy1	z0; Vz1	
?t1 w[x]; !t1 Vx1			t1: w[x]w[z] Vx1y2z2
?t2 w[x]; !t2 Vx1	?t2 w[y]; !t2 Vy1		t2: w[x]w[y] Vx1y1
?t2 Vx1y1	?t3 w[y]; !t3 Vy1	?t3 w[z]; !t3 Vz1	t3: w[y]w[z] Vy1z1
	?t3 Vy1z1	?t3 Vy1z1	
	?t2 Vx1y1	frame z[3]; Vy2z2	
		?t1 w[z]; !t1 Vy2z2	
?t1 Vx1y2z2		?t1 Vx1y2z2	
2 < 1	2 < 3	3 < 1	
t1 Vx1y2z2	t2 Vx1y1		
t2 Vx1y1z2	t3 Vx1y1z1		

?	receive
!	send
t3	txn 3
V	Vector Clock
x0	x at vsn 0
r[x0]	read of x
w[x]	write of x

- Hardest part of Paxos is garbage collection

- Hardest part of Paxos is garbage collection
- Need additional messages to determine when Paxos instances can be deleted

- Hardest part of Paxos is garbage collection
- Need additional messages to determine when Paxos instances can be deleted
- We can use these to also express:
  You will never see any of these vector clock elems again

- Hardest part of Paxos is garbage collection
- Need additional messages to determine when Paxos instances can be deleted
- We can use these to also express:
  You will never see any of these vector clock elems again
- Therefore we can remove matching elems from vector clocks!
- Many more details elided!

Vector clocks capture dependencies and causal relationship between transactions

Vector clocks capture dependencies and causal relationship between transactions

Plus we always add transactions into the youngest frame

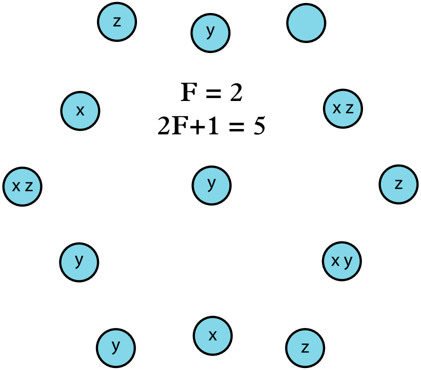
Vector clocks capture dependencies and causal relationship between transactions
Plus we always add transactions into the youngest frame
Which gets us Strong Serializability

No leader, so no potential bottleneck

No leader, so no potential bottleneck
No wall clocks, so no issues with clock skews

No leader, so no potential bottleneck
No wall clocks, so no issues with clock skews
Can separate F from cluster size,

No leader, so no potential bottleneck
No wall clocks, so no issues with clock skews
Can separate F from cluster size,
Which gets us horizontal scalability


z
y
F = 2
2F+1 = 5
x
x z
x z
y
z
y
x y
y
x
z

- Interval Tree Clocks - Paulo Sérgio Almeida, Carlos Baquero, Victor Fonte
- Highly available transactions: Virtues and limitations - Bailis et al
- Coordination avoidance in database systems - Bailis et al
- k-dependency vectors: A scalable causality-tracking protocol - Baldoni, Melideo
- Multiversion concurrency control-theory and algorithms - Bernstein, Goodman
- Serializable isolation for snapshot databases - Cahill, Röhm, Fekete
- Paxos made live: an engineering perspective - Chandra, Griesemer, Redstone

- Consensus on transaction commit - Gray, Lamport
- Spanner: Google's globally distributed database - Corbett et al
- Faster generation of shorthand universal cycles for permutations - Holroyd, Ruskey, Williams
- s-Overlap Cycles for Permutations - Horan, Hurlbert
- Universal cycles of k-subsets and k-permutations - Jackson
- Zab: High-performance broadcast for primary-backup systems - Junqueira, Reed, Serafini
- Time, clocks, and the ordering of events in a distributed system - Lamport

References (3)

- The part-time parliament - Lamport
- Paxos made simple - Lamport
- Consistency, Availability, and Convergence - Mahajan, Alvisi, Dahlin
- Notes on Theory of Distributed Systems - Aspnes
- In search of an understandable consensus algorithm - Ongaro, Ousterhaut
- Perfect Consistent Hashing - Sackman
- The case for determinism in database systems - Thomson, Abadi

GoshawkDB

Distributed databases are FUN!

https://goshawkdb.io/