

# Secure Distributed Programming

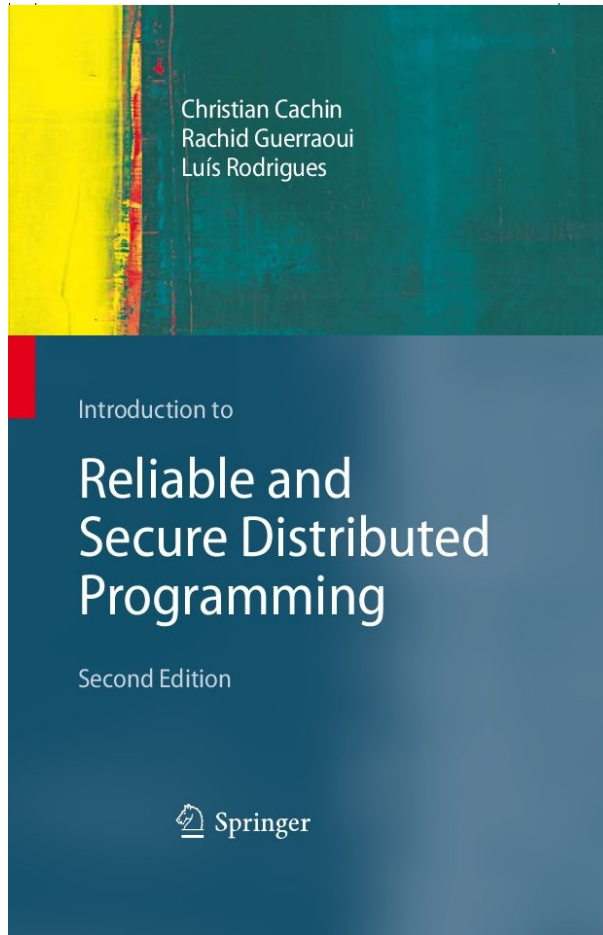
Christian Cachin\*  
Rachid Guerraoui  
Luís Rodrigues

Tutorial at CCS 2011

# *A play in three acts*

- Abstractions and protocols for
  - Reliable broadcast
  - Shared memory
  - Consensus
- In asynchronous distributed systems
- With processes subject to
  - Crash failures
  - Malicious attacks / Byzantine failures

# Background



## Introduction to Reliable and Secure Distributed Programming

- C. Cachin, R. Guerraoui, L. Rodrigues
- 2nd ed. of "Introduction to Reliable Distributed Programming" (Springer, 2011)
- The new content covers Byzantine failures

Web: [www.distributedprogramming.net](http://www.distributedprogramming.net)

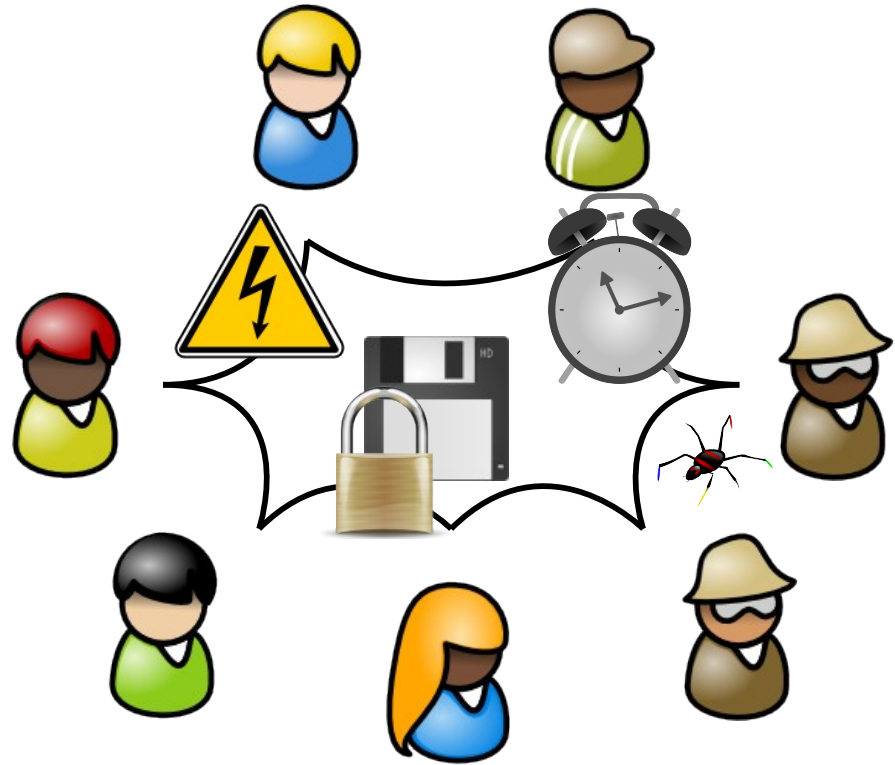
---

---

# *Distributed systems*

- Basic abstractions

- Processes
- Links
- Timing models
- Cryptography



# **Prologue**

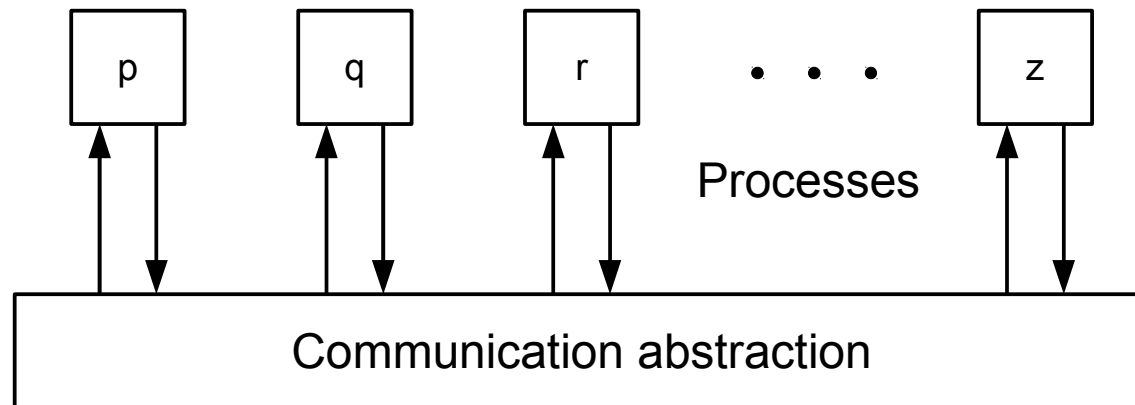
## **Models and assumptions**



# *Programming abstractions*

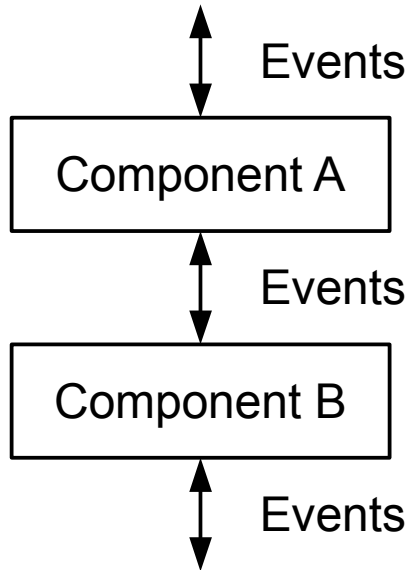
- **Sequential programming**
    - Array, record, list ...
  - **Concurrent programming**
    - Thread, semaphore, monitor ...
  - **Distributed programming**
    - Reliable broadcast
    - Shared memory
    - Consensus
    - Atomic commit
    - ...
- 
-

# *Distributed programming abstractions*



- Coordination among **N** identical processes
  - Processes are also called replicas
- Processes jointly implement application
  - Need coordination

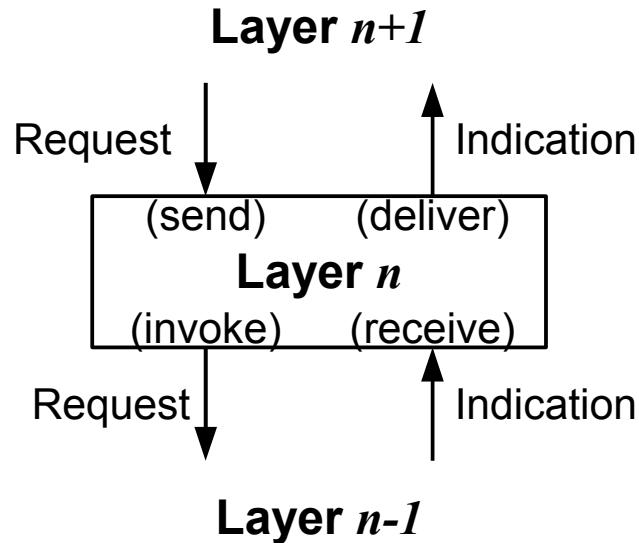
# *Layered modular architecture*



- Every process consists of **modules**
  - Modules may exist in multiple instances
  - Every instance has a unique identifier
- Modules communicate through **events**



# Programming with events

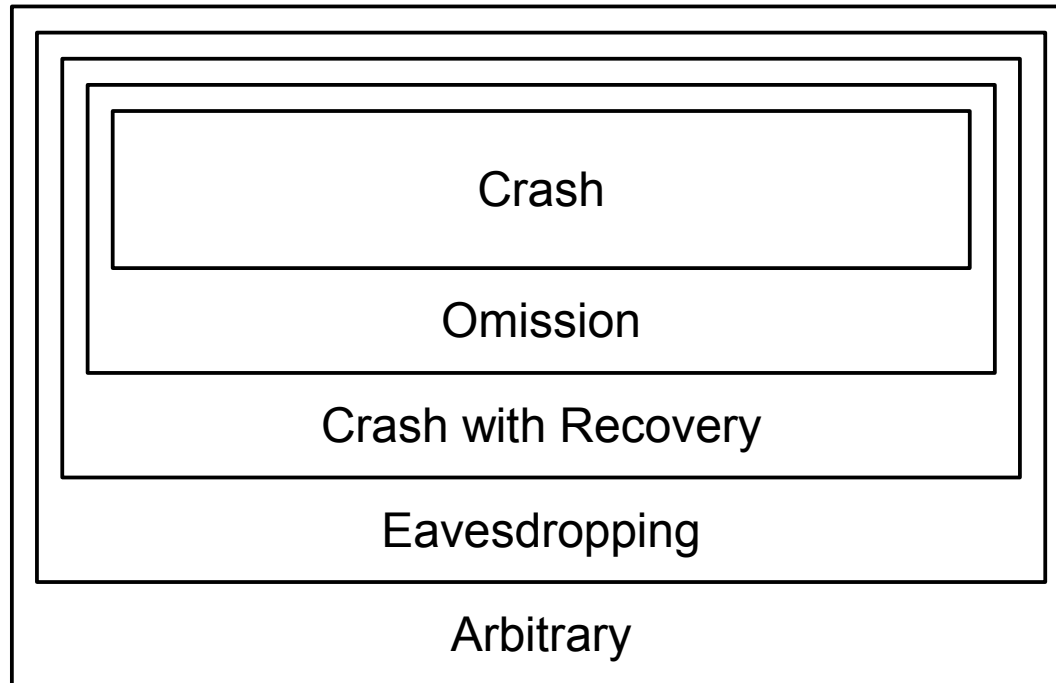


- Modules are arranged in layers of a stack
- Asynchronous events represent communication or control flow
  - **Request** events flow downward
  - **Indication** events flow upward

# Processes

- System with **N** processes  $\Pi = \{p, q, r \dots\}$
  - Processes know each other
  - Every process consists of a set of modules and interacts through events
  - Reactive programming model  
**upon event** <mod, Event | att<sub>1</sub>, att<sub>2</sub> ...> **do**  
do something;  
**trigger** <mod', Event' | att'<sub>1</sub>, att'<sub>2</sub> ...>;
- 
-

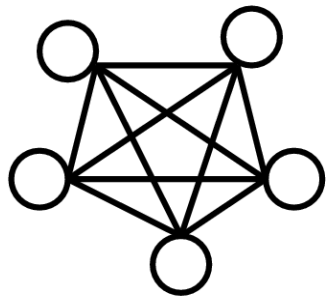
# Process failures



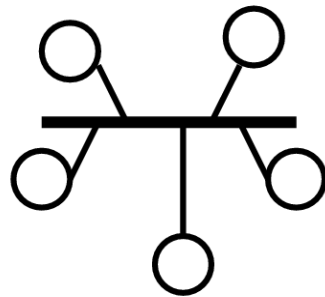
- In this tutorial, we consider only:
  - **Crash failures**
    - Failed process stops executing steps
  - **Arbitrary or "Byzantine" failures**
    - Failed process behaves arbitrarily and adversarially
    - May not break cryptographic primitives

# Links

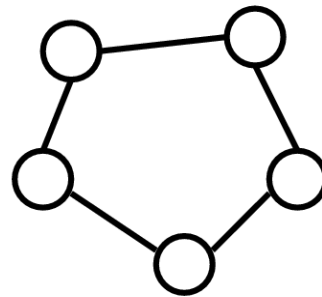
- Logically every process may communicate with every other process: (a)
- Physical implementation may differ: (b)-(d)



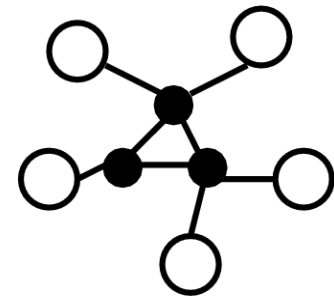
(a)



(b)



(c)



(d)



# *Perfect Point-to-point Links (pl)*

- Events

- Request  $\langle pl, \text{Send} \mid q, m \rangle$ 
  - Sends a message  $m$  to process  $q$
- Indication  $\langle pl, \text{Deliver} \mid p, m \rangle$ 
  - Delivers a message  $m$  from sender  $p$

- Properties

- PL1 (Reliability): If a correct sends message  $m$  to a correct process  $q$ , then  $q$  eventually delivers  $m$ .
  - PL2 (No duplication): No message is delivered more than once.
  - PL3 (No creation): If a process delivers a message  $m$  with sender  $p$ , then  $s$  has sent  $m$ .
- 
-

# *Time*

- Most algorithms shown here are asynchronous
  - No bounds on message transmission time or process execution time
- Some algorithms use an abstraction of time
  - Failure detector
  - Leadership detector



# ***Cryptographic primitives***

- Dual goals of cryptography
- Confidentiality
  - Encryption (not relevant here)
- Integrity
  - Hash functions
  - Message authentication codes (MAC)
  - Digital signatures



# Hash functions

- Cryptographic hash function **H** maps inputs of arbitrary length to a short unique tag
  - **Collision-freeness**: No process can find distinct values **x** and **x'** such that  **$H(x) = H(x')$**
  - Formally, implemented by a distributed oracle
    - Maintains list **L** of inputs given to **H** so far
    - **upon** invocation  **$H(x)$** 
      - **if  $x \in L$ , then** append **x** to **L**
      - **return** index of **x** in **L**
    - Practical hash functions have more properties not modeled here
- 
-



# Message-authentication codes

- A MAC authenticates data between two processes (messages from sender to receiver)
  - Formally, given by a distributed oracle
    - Maintains set  $A$  of strings authenticated so far
    - **upon** invocation `authenticate(p, q, m)` // only by  $p$ 
      - pick authenticator  $a$ , add  $(p,q,m,a)$  to  $A$
      - **return**  $a$
    - **upon** invocation `verifyauth(q, p, m, a)` // only by  $q$ 
      - **if**  $(p,q,m,a) \in A$  **then**
        - **return** TRUE
      - **else**
        - **return** FALSE
    - Implemented with shared secret key and hash functions
- 
-

# Digital signatures

- A digital signature scheme authenticates data with public verification
  - Formally, given by a distributed oracle
    - Maintains set **S** of strings signed so far
    - **upon** invocation **sign(p, m)** // only by **p**
      - pick signature **s**, add **(p,m,s)** to **S**
      - **return s**
    - **upon** invocation **verifysig(q, m, s)** // by anyone
      - **if (q,m,s) ∈ S then**
        - **return TRUE**
      - **else**
        - **return FALSE**
    - Implemented from public-key cryptosystems
    - Authenticity can be relayed by untrusted process
- 
-

# **Act I**

## **Reliable broadcast**



# Broadcast

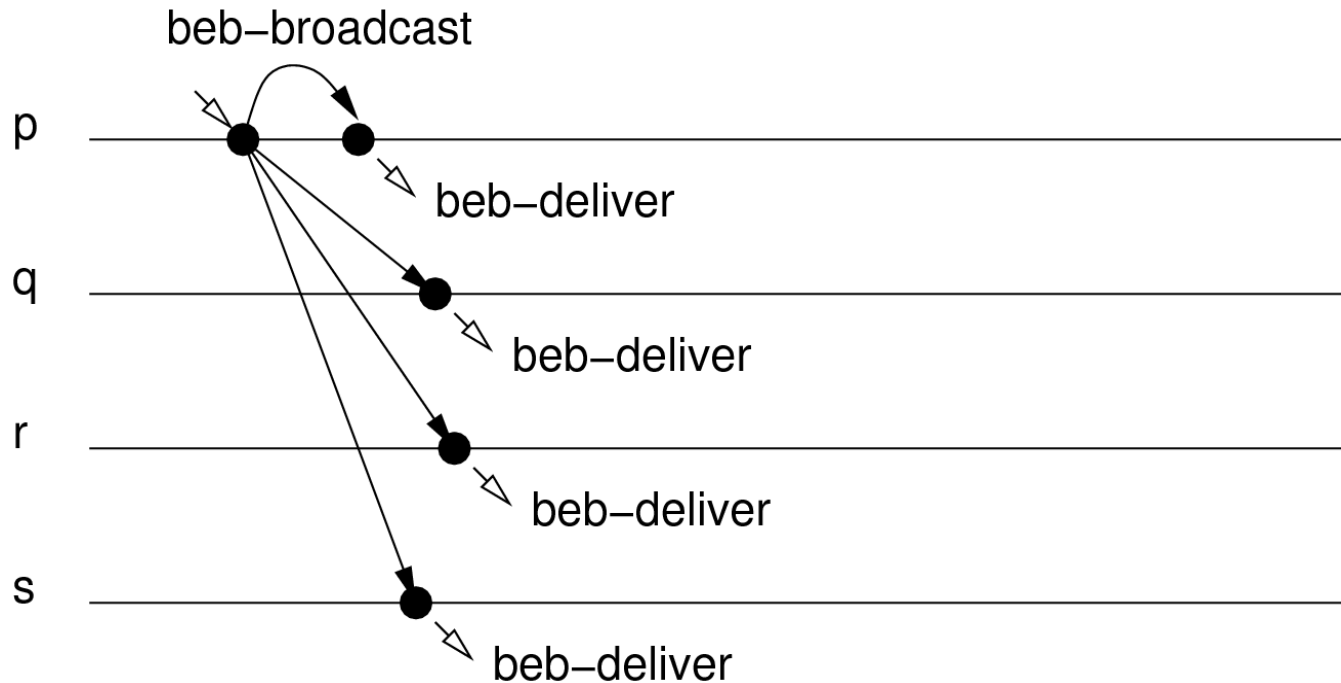
- Broadcast is a basic primitive to disseminate information
    - Processes in the group send messages
    - All processes should receive or "deliver" the messages
  - Reliable broadcast
    - Guarantees that messages are delivered to all processes consistently
    - Agreement on the delivered messages
    - No ordering among delivered messages
- 
-

# Best-Effort Broadcast (*beb*)

- Events
    - Request  $\langle \text{beb}, \text{Broadcast} \mid m \rangle$ 
      - Broadcasts a message  $m$  to all processes
    - Indication  $\langle \text{beb}, \text{Deliver} \mid p, m \rangle$ 
      - Delivers a message  $m$  from sender  $p$
  - Properties
    - BEB1 (Validity): If a correct process broadcasts  $m$ , then every correct process eventually delivers  $m$ .
    - BEB2 (No duplication): No message is delivered more than once.
    - BEB3 (No creation): If a process delivers a message  $m$  with sender  $p$ , then  $p$  has broadcast  $m$ .
  - Offers no "reliability" when a process fails
- 
-

# Best-effort broadcast protocol

- Sender sends message **m** to all processes over point-to-point links
- **Not reliable**



# ***Uniform Reliable Broadcast (urb)***

- Events
    - Request  $\langle \text{urb}, \text{Broadcast} \mid m \rangle$ 
      - Broadcasts a message  $m$  to all processes
    - Indication  $\langle \text{urb}, \text{Deliver} \mid p, m \rangle$ 
      - Delivers a message  $m$  from sender  $p$
  - Properties
    - RB1 (Validity) = BEB1
    - RB2 (No duplication) = BEB2
    - RB3 (No creation) = BEB3
    - RB4 (Uniform agreement): If some process\* delivers a message  $m$ , then every correct process eventually delivers  $m$ .

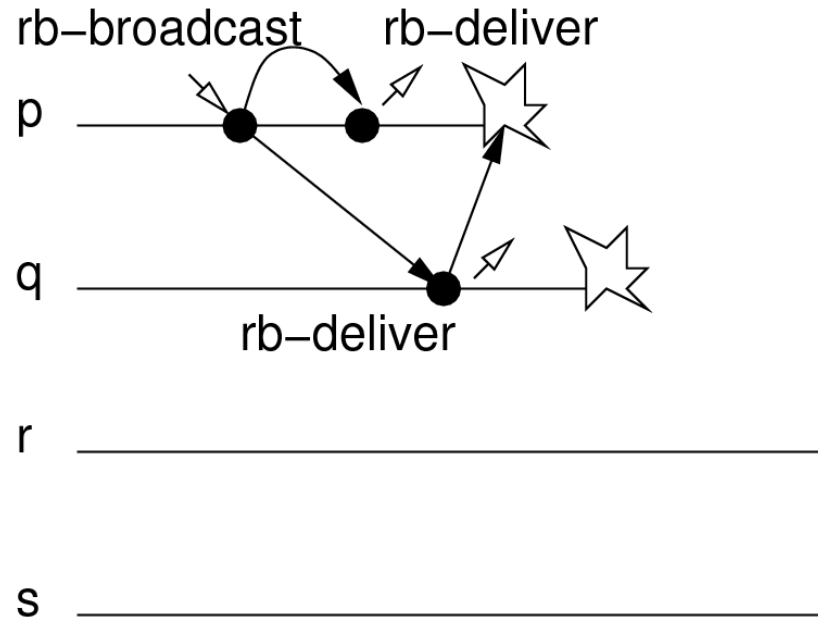
\* whether process is correct or faulty!
- 
-

# *Why uniform agreement?*

- A process **p** delivers a message **m** and crashes later; still every correct process must deliver **m**.
  - The notion of regular reliable broadcast requires this only when **p** is correct (= never fails).
  - When **p** may influence an application or the environment before it crashes, other processes will also deliver message, consistent with **p**.
- 
-



# Regular reliable broadcast



- Example of reliable but non-uniform execution
- Process **p** delivers **m**
- No other process delivers **m**

# Majority-Ack Uniform Reliable Broadcast

Implements **urb**, uses **beb** ( $N > 2f$ )

delivered :=  $\emptyset$ ; pending :=  $\emptyset$ ;  $\forall m : \text{ack}[m] := \emptyset$

```
upon <urb, Broadcast | m> do  
  pending := pending  $\cup$  {(self,m)}  
  for q $\in$  $\Pi$  do trigger <beb, Broadcast | [DATA, self, m]>
```

```
upon <beb, Deliver | p, [DATA, s, m]> do  
  ack[m] := ack[m]  $\cup$  {p}  
  if (s,m)  $\notin$  pending then  
    pending := pending  $\cup$  {(s,m)}  
    for q $\in$  $\Pi$  do  
      trigger <beb, Broadcast | [DATA, self, m]>
```

(...)

---

---

# Majority-Ack Uniform Reliable Broadcast

(...)

**upon**  $\exists (s,m) \in \text{pending} : m \notin \text{delivered} \wedge \#\text{ack}[m] > N/2$  **do**  
     $\text{delivered} := \text{delivered} \cup \{m\}$   
    **trigger**  $\langle \text{urb}, \text{Deliver} \mid s, m \rangle$

- Delivers message **m** only after **m** has been relayed by a majority of processes
- Every majority contains at least one correct process



# ***Byzantine reliable broadcasts***

- Almost the same primitive: needs to reach agreement on delivered messages
  - Byzantine sender may cause processes to deliver different message content for the "same" message
  - **How to identify a message?**
- 
-

# Messages not self-explaining

- Important change from model with crashes
    - With crash failures, a reliable broadcast module delivers **many** messages
      - Messages are unique and identified only by their content
    - With Byzantine processes, this is problematic
      - Since messages are not ordered, and Byz. sender may send any message, application may become confused
      - Ex.: application broadcasts message  $[l,m]$ , containing a payload  $m$  and a label  $l$ ; faulty sender may cause  $p$  to deliver  $[l,m]$  first and  $q$  to deliver  $[l,m']$  first, with  $m \neq m'$
  - A Byzantine reliable broadcast instance
    - Corresponds to **one** delivered message
    - A priori declares a **sender** process for the instance
- 
-

# *Authenticated communication primitives*

- Recall modules in model with crash failures
    - Perfect Links (pl)
    - Best-effort Broadcast (beb) modules
  - Authenticated versions can be defined that tolerate network subject to attacks
    - Authenticated Perfect Links (al)
    - Authenticated Best-effort Broadcast (abeb)
    - Implemented using cryptographic authentication (MACs or digital signatures)
- 
-

# ***Byzantine broadcast variants***

- Byzantine consistent broadcast
- Byzantine reliable broadcast



# Byzantine Consistent Bc. (bcb)

- Events
    - Request  $\langle \text{bcb}, \text{Broadcast} \mid m \rangle$ 
      - Broadcasts a message  $m$  to all processes
    - Indication  $\langle \text{bcb}, \text{Deliver} \mid p, m \rangle$ 
      - Delivers a message  $m$  from sender  $p$
  - Properties
    - BCB1 (Validity) = BEB1
    - BCB2 (No duplication): Every correct process delivers at most one message
    - BCB3 (Integrity): If a correct process delivers  $m$  with sender  $p$ , and  $p$  correct, then  $p$  has broadcast  $m$ .
    - (...)
- 
-



# ***Byzantine Consistent Bc. (bcb)*** ***(cont.)***

(... Properties)

- **BCB4 (Consistency)**: If a correct process delivers message **m** and another correct process delivers message **m'**, then **m=m'**.
- Note: some correct process may not deliver any message (agreement is not needed)



# ***Auth. Echo Broadcast***

Implements **bcb**, uses **abeb**, with sender **s** ( $N > 3f$ ) [ST87]

```
upon <bcb, Broadcast | m> do  
  trigger <abeb, Broadcast | [SEND, m]>
```

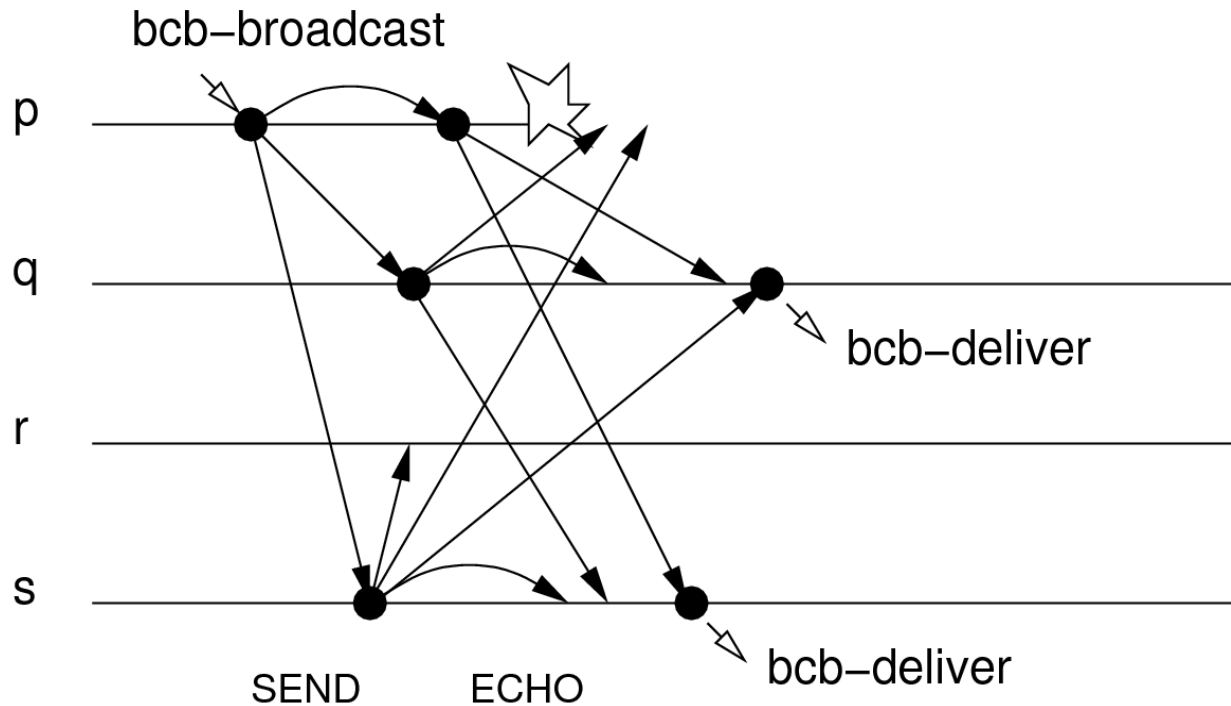
```
upon <abeb, Deliver | s, [SEND, m]> do  
  trigger <abeb, Broadcast | [ECHO, m]>
```

```
upon <abeb, Deliver | p, [ECHO, m]> do  
  echo[p] := m  
  if  $\exists m : \#\{p \mid \text{echo}[p]=m\} > (N+f)/2$  then  
    trigger <bcb, Deliver | s, m>
```

```
// code to prevent duplicate execution is omitted
```



# Example



- Faulty sender **p**
- Processes **q** and **s** bcb-deliver the message
- Process **r** does not deliver any message
- $O(n^2)$  messages;  $O(n^2 |m|)$  communication

# Using Byzantine quorums

- System of  $N > 3f$  processes,  $f$  are faulty
  - Every subset with size strictly larger than  $(N+f)/2$  processes is a Byzantine quorum (B.Q.)
    - Every B.Q. has more than  $(N-f)/2$  correct processes
    - Two distinct B.Q. together contain more than  $N-f$  correct pr.
    - Thus, every two B.Q. overlap in some correct pr.
      - This correct process has abeb-broadcast the same message [ECHO,  $m$ ] to all processes
  - The collection of all Byzantine quorums is a quorum system
- 
-

# ***Byzantine Reliable Bc. (brb)***

- Events
    - Request  $\langle \text{brb, Broadcast} \mid m \rangle$
    - Indication  $\langle \text{brb, Deliver} \mid p, m \rangle$
  - Properties
    - BRB1 (Validity) = BCB1
    - BRB2 (No duplication) = BCB2
    - BRB3 (Integrity) = BCB3
    - BRB4 (Consistency) = BCB4
    - BRB5 (Totality): If some correct process delivers a message, then every correct process eventually delivers a message
  - Either all or none of the correct processes deliver the message
- 
-

# ***Auth. Double-Echo Broadcast***

Implements **brb**, uses **abeb**, with sender **s** ( $N > 3f$ ) [Bra87]

sentready := FALSE

**upon** <**brb**, Broadcast | **m**> **do**

**trigger** <**abeb**, Broadcast | [SEND, **m**]>

**upon** <**abeb**, Deliver | **s**, [SEND, **m**]> **do**

**trigger** <**abeb**, Broadcast | [ECHO, **m**]>

**upon** <**abeb**, Deliver | **p**, [ECHO, **m**]> **do**

    echo[**p**] := **m**

**if**  $\exists m : \#\{p \mid \text{echo}[p]=m\} > (N+f)/2 \wedge \neg \text{sentready}$  **then**

        sentready := TRUE

**trigger** <**abeb**, Broadcast | [READY, **m**]>

(...)

---

---

# ***Auth. Double-Echo Broadcast***

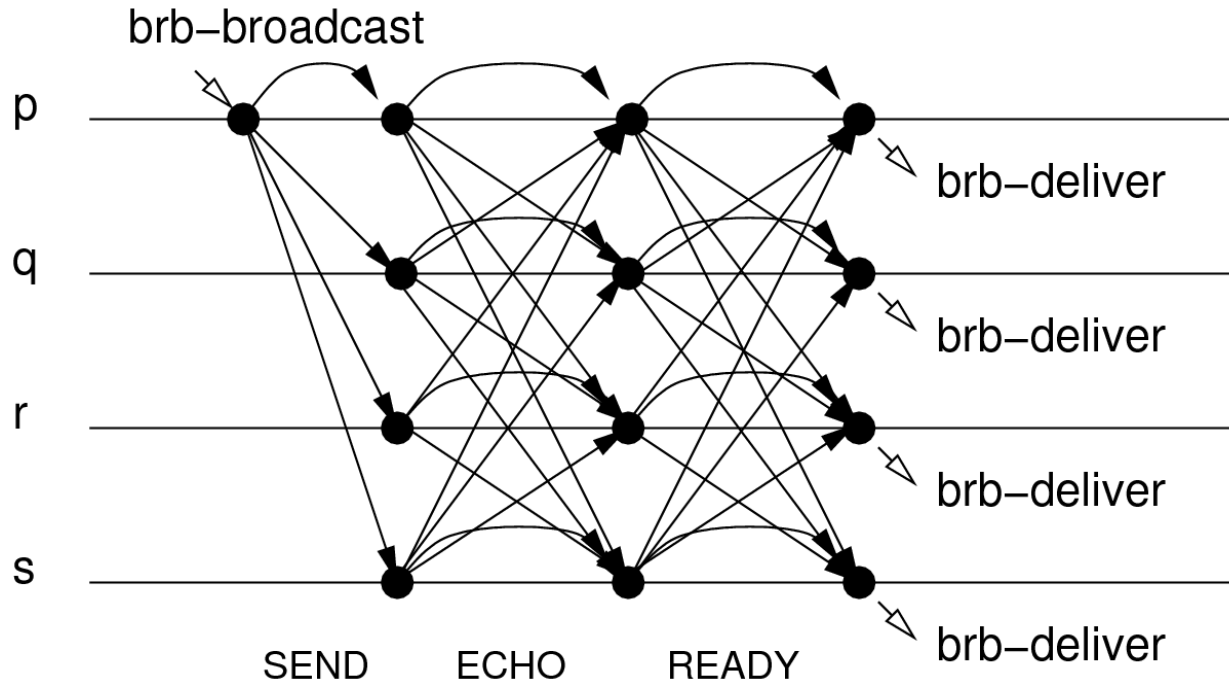
(...)

```
upon <abeb, Deliver | p, [READY, m]> do  
  ready[p] := m  
  if  $\exists m : \#\{p \mid \text{ready}[p]=m\} > f \wedge \neg \text{sentready}$  then  
    // amplification of READY messages  
    sentready := TRUE  
    trigger <abeb, Broadcast | [READY, m]>  
  else if  $\exists m : \#\{p \mid \text{ready}[p]=m\} > 2f$  then  
    trigger <brb, Deliver| s, m>
```

// again, some code to prevent duplicate execution is omitted



# Example



- Amplification from  $f+1$  to  $2f+1$  READY messages ensures totality
  - All or none of the correct processes deliver message
- $O(n^2)$  messages;  $O(n^2 |m|)$  communication



# ***Byzantine Broadcast Channel***

- Combines many one-message broadcast instances
  - Every message delivered together with a unique label
    - Consistency and totality hold for each label
  - Implemented from multiple "parallel" instances of Byzantine broadcasts
  - Two variants
    - Consistent Channel
    - Reliable Channel
- 
-

# **Act II**

## **Shared memory**



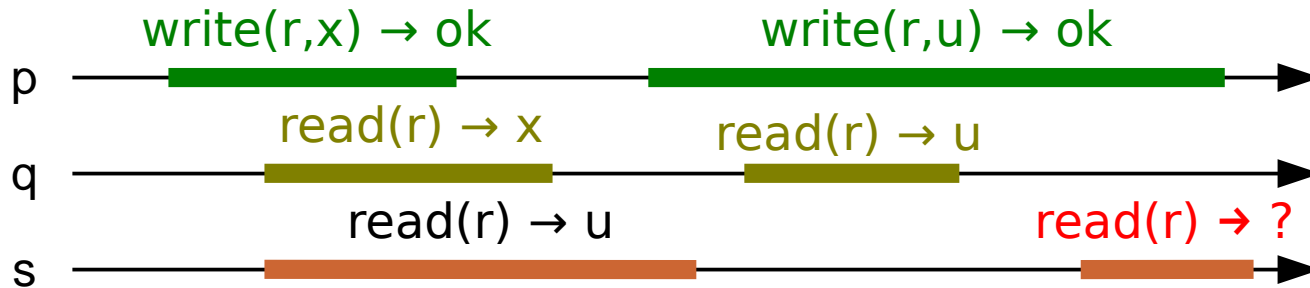
# *Operations on shared memory*

- Memory abstraction is a register
  - Two operations: **read and write**
  - Operations restricted to certain processes
    - 1 writer or N writers
    - 1 reader or N readers
    - **(W,R)-register has W writers and R readers**
- 
-

# Concurrent operations

- Operations take time, defined by two events at a process: **invocation** and **completion**
  - **Write( $r, v$ )  $\rightarrow$  ok**
    - Writes value **v** to register instance **r**
  - **Read( $r$ )  $\rightarrow v$** 
    - Reads from register instance **r** and returns value **v**
  - Operation **o** **precedes** **o'** whenever completion of **o** occurs before invocation of **o'**
  - Otherwise, **o** and **o'** are **concurrent**
- 
-

# Semantics of memory ops.



**Safe:** Every **read** not concurrent with a **write** returns the most recently *written* value.

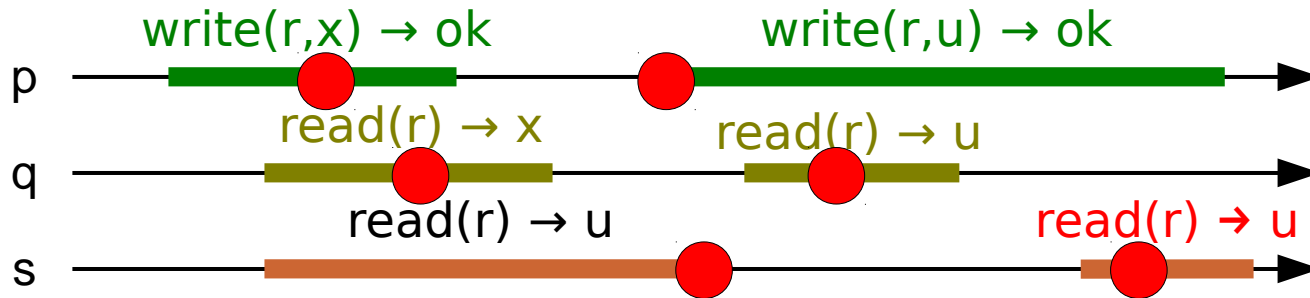
**Regular:** *Safe* & any **read** concurrent with a **write** returns either the most recently *written* value or the concurrently *written* value: **process s may read x or u.**

**Atomic:** *Regular* & all **read** and **write** operations occur atomically ( = linearizable): **process s must read u.**

---

---

# Linearizability



- Every operations appears to execute **atomically** at its linearization point ● which lies in real time between the invocation and the completion

# *(1,N) Regular Register (onrr)*

- Events

- Request  $\langle \text{onrr}, \text{Read} \rangle$ 
  - Invokes a read operation on the register
- Request  $\langle \text{onrr}, \text{Write} \mid v \rangle$ 
  - Invokes a write operation with value  $v$
- Indication  $\langle \text{onrr}, \text{ReadReturn} \mid v \rangle$ 
  - Completes a read operation, returning value  $v$
- Indication  $\langle \text{onrr}, \text{WriteReturn} \rangle$ 
  - Completes a write operation

- Properties

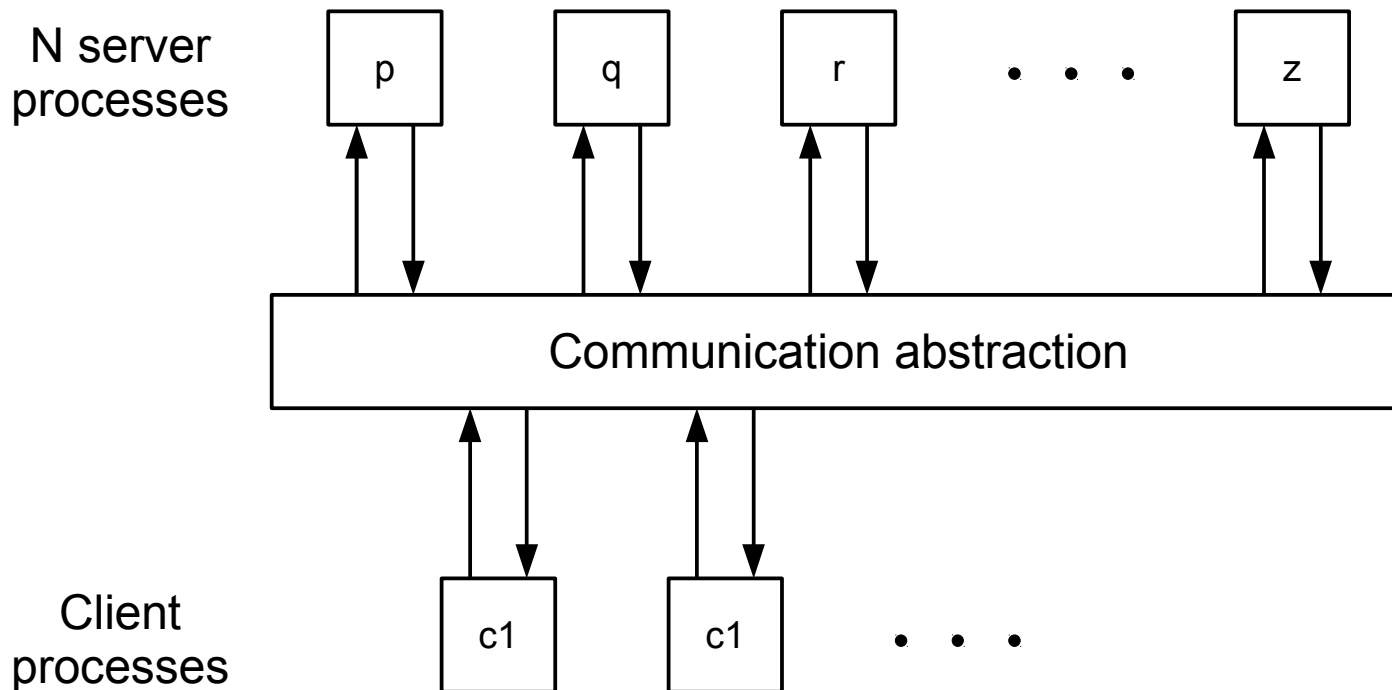
- **ONRR1 (Liveness)**: If a correct process invokes an operation, then the operation eventually completes.
  - **ONRR1 (Validity)**: A read returns the last value written or the\* value written concurrently.
    - \*Only one process can possibly write.
- 
-

# *Implementations of registers*

- **From other (simpler, unreliable) registers**
  - Multi-valued from binary registers
  - (N,N) and (1,N) from (1,1) registers
  - Regular registers from safe registers
  - Atomic registers from regular registers
  - ...
- **From replicated (unreliable) processes**
  - Considered here
  - Replica processes may fail
    - Crash failures
    - Byzantine failures



# Client-server model



- **Clients** and **servers** are usually separate
- For simplicity, we model them here as one group of N processes
  - Processes have **dual role** as clients and servers

# Majority-Voting Reg. Register

Implements **onrr**, uses **pl**, **beb** ( $N > 2f$ )

$(ts, val) := (0, \perp)$ ;  $wts := 0$ ;  $rid := 0$

**upon**  $\langle$ **onrr**, Write |  $v$  $\rangle$  **do**  
     $wts := wts + 1$   
     $acklist := [\perp]^N$   
    **trigger**  $\langle$ **beb**, Broadcast | [WRITE,  $wts$ ,  $v$ ] $\rangle$

**upon**  $\langle$ **beb**, Deliver |  $p$ , [WRITE,  $ts'$ ,  $v'$ ] $\rangle$  **do**  
    **if**  $ts' > ts$  **then**  
         $(ts, val) := (ts', v')$   
    **trigger**  $\langle$ **pl**, Send |  $p$ , [ACK,  $ts'$ ] $\rangle$

**upon**  $\langle$ **pl**, Deliver |  $q$ , [ACK,  $wts$ ] $\rangle$  **do**  
     $acklist[q] := 1$   
    **if**  $\#(acklist) > N/2$  **then**  
        **trigger**  $\langle$ **onrr**, WriteReturn $\rangle$

(...)

---

# Majority-Voting Reg. Register

(...)

**upon** <onrr, Read> **do**

rid := rid + 1

readlist := [ $\perp$ ]<sup>N</sup>

**trigger** <beb, Broadcast | [READ, rid]>

**upon** <beb, Deliver | p, [READ, r]> **do**

**trigger** <pl, Send | p, [VALUE, r, ts, val]>

**upon** <pl, Deliver | q, [VALUE, rid, ts', v']> **do**

readlist[q] := (ts', v')

**if** #(readlist) > N/2 **then**

v := highestval(readlist) // value with highest ts

**trigger** <onrr, ReadReturn | v>

- Validity: every two operations access one common correct process
- 
-

# *Registers in Byzantine model*

- Up to  $f$  processes may be (Byzantine) faulty, including reader
- Writer process is always correct
- Specification of
  - $(1,N)$  safe Byzantine register (bonsr) and
  - $(1,N)$  regular Byzantine register (bonrr)directly follows from  $(1,N)$  regular register

# *Implementations*

- Algorithms must eliminate wrong values returned by Byzantine processes
  - Two approaches for elimination
    - Masking by sufficiently many correct values
      - Alg. "Masking Quorum" for Byzantine safe register
    - Authentication of correct values with digital signatures
      - Alg. "Authenticated-Data" for Byzantine regular register
- 
-

# Byzantine Masking Quorum

Implements **bonsr**, uses **al**, **abeb** ( $N > 4f$ ), writer is **w**

(ts, val) := (0,  $\perp$ ); wts := 0; rid := 0 // Differences are in this color

**upon** <**bonsr**, Write | v> **do**  
    wts := wts + 1  
    acklist := [ $\perp$ ]<sup>N</sup>  
    **trigger** <**abeb**, Broadcast | [WRITE, wts, v]>

**upon** <**abeb**, Deliver | w, [WRITE, ts', v']> **do**  
    **if** ts' > ts **then**  
        (ts, val) := (ts', v')  
    **trigger** <**al**, Send | w, [ACK, ts']>

**upon** <**al**, Deliver | q, [ACK, wts]> **do**  
    acklist[q] := 1  
    **if** #(acklist) >  $(N+2f)/2$  **then**  
        **trigger** <**bonsr**, WriteReturn>

(...)

---

# Byzantine Masking Quorum

(...)

**upon** <bonsr, Read> **do**

rid := rid + 1

readlist :=  $[\perp]^N$

**trigger** <abeb, Broadcast | [READ, rid]>

**upon** <abeb, Deliver | p, [READ, r]> **do**

**trigger** <al, Send | p, [VALUE, r, ts, val]>

**upon** <al, Deliver | q, [VALUE, rid, ts', v']> **do**

readlist[q] := (ts', v')

**if** #(readlist) >  $(N+2f)/2$  **then**

v := byz-highestval(readlist) // filter and extract value

**trigger** <bonsrr, ReadReturn | v>

- byz-highestval()
  - eliminates all values occurring f or fewer times
  - returns survivor value with highest timestamp
  - or -- special value  $\perp$  if no such value exists

# Comments

- Alg. Byzantine Masking Quorum may return  $\perp$ 
    - Implements safe register on domain with  $\{\perp\}$
  - Without concurrent write operation
    - Last write op. has **touched** more than  $(N+2f)/2$  pr.
      - Among them, more than  $(N+2f)/2 - f$  are correct
      - Less than  $(N-2f)/2$  correct processes are **untouched**
    - Read op. obtains value from more than  $(N+2f)/2$  pr.
      - Up to  $f$  may be from Byzantine pr.
      - Less than  $(N-2f)/2$  are from **untouched** correct pr.
      - Since  $N > 4f$ , it follows  $(N+2f)/2 - f - (N-2f)/2 = f$ .
      - Hence, strictly more than  $f$  values are from correct pr. and contain **last-written timestamp/value pair**
- 
-



# Auth.-Data Byzantine Quorum

Implements **bonrr**, uses **al**, **abeb**, signatures ( $N > 3f$ ), writer is **w**

$(ts, val, s) := (0, \perp, \perp)$ ;  $wts := 0$ ;  $rid := 0$  // Differences are in this color

```
upon <bonrr, Write | v> do  
   $wts := wts + 1$ ;  $s := \text{sign}(w, \text{WRITE} || w || wts || v)$   
   $acklist := [\perp]^N$   
  trigger <abeb, Broadcast | [WRITE, wts, v, s]>
```

```
upon <abeb, Deliver | w, [WRITE, ts', v', s']> do  
  if  $ts' > ts$  then  
     $(ts, val, s) := (ts', v', s')$   
  trigger <al, Send | w, [ACK, ts']>
```

```
upon <al, Deliver | q, [ACK, wts]> do  
   $acklist[q] := 1$   
  if  $\#(acklist) > (N+f)/2$  then  
    trigger <bonsr, WriteReturn>
```

(...)

---

# ***Auth.-Data Byzantine Quorum***

(...)

**upon** <bonrr, Read> **do**

rid := rid + 1

readlist := [ $\perp$ ]<sup>N</sup>

**trigger** <abeb, Broadcast | [READ, rid]>

**upon** <abeb, Deliver | p, [READ, r]> **do**

**trigger** <al, Send | p, [VALUE, r, ts, val, s]>

**upon** <al, Deliver | q, [VALUE, rid, ts', v', s']> **do**

**if** **verifysig**(w, WRITE||w||ts' ||v', s') **then**

readlist[q] := (ts', v')

**if** #(readlist) > (N+f)/2 **then**

v := **highestval**(readlist) // value with highest ts

**trigger** <bonrr, ReadReturn | v>

---

---

# Comments

- Alg. Authenticated-Data Byz. Quorum uses
  - Digital signatures issued by writer
  - Byzantine quorums
- Otherwise, exactly the same as the Majority Quorum algorithm
  - Signatures **authenticate** the value
  - Signatures **bind value to timestamp**

**Act III**

**Consensus**



# Consensus

- Processes propose values and have to **agree on one decision value** among the proposed values
- Consensus is a key abstraction for solving many other problems in fault-tolerant distributed systems
  - Total-order broadcast
  - Non-blocking atomic commit
  - Replicated services
  - ...

# *Uniform Consensus (uc)*

- Events
  - Request  $\langle uc, Propose \mid v \rangle$ 
    - Proposes value  $v$  for consensus
  - Indication  $\langle uc, Decide \mid v \rangle$ 
    - Outputs a decided value  $v$  of consensus
- Properties
  - UC1 (Termination): Every correct process eventually decides.
  - UC2 (Validity): Any decided value has been proposed by some process.
  - UC3 (Integrity): No process decides twice.
  - UC4 (Uniform Agreement): No two processes\* decide differently.

\* whether correct or faulty

---

---

# Weak Byzantine Cons. (wbc)

- Events

- Request  $\langle \text{wbc}, \text{Propose} \mid v \rangle$ 
  - Proposes value  $v$  for consensus
- Indication  $\langle \text{wbc}, \text{Decide} \mid v \rangle$ 
  - Outputs a decided value  $v$  of consensus

- Properties

- WBC1 (Termination) = UC1
  - WBC2 (Weak Validity): Suppose all processes are correct: if all propose  $v$ , then a process may only decide  $v$ ; if a process decides  $v$ , then  $v$  was proposed by some process.
  - WBC3 (Integrity): No correct process decides twice.
  - WBC4 (Agreement): No two correct processes decide differently.
- 
-

# *Implementing consensus*

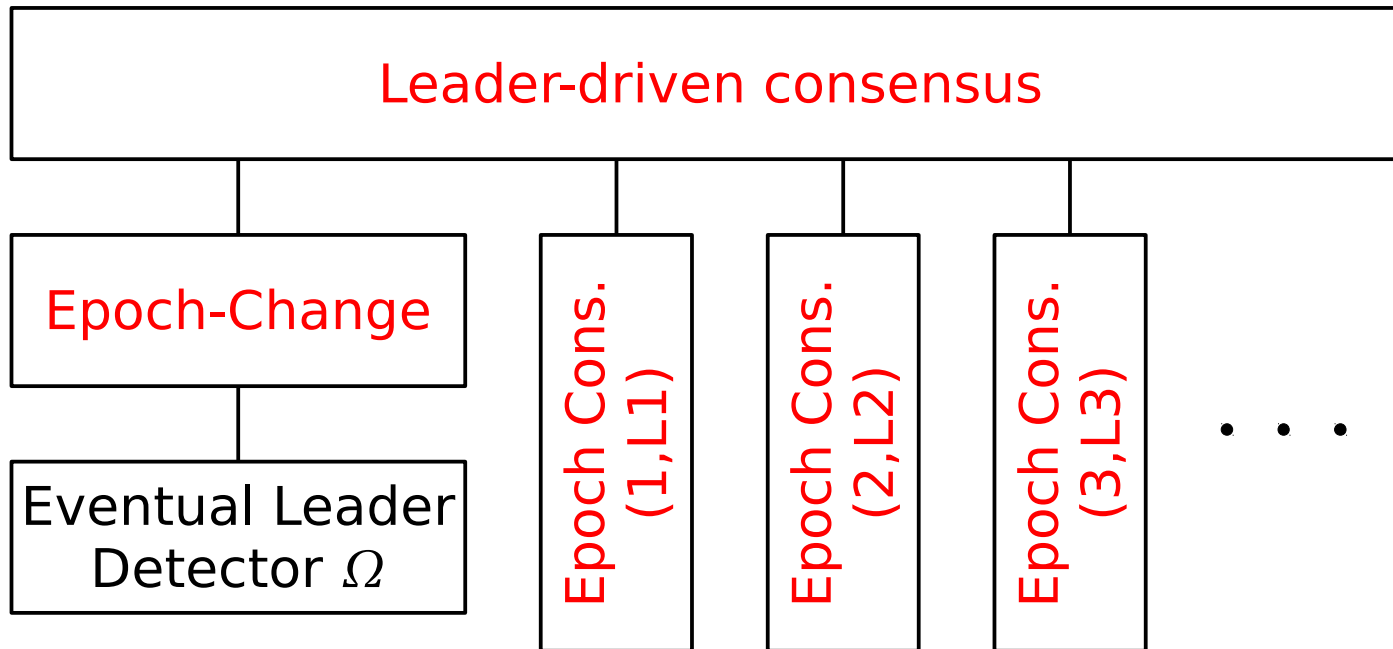
- In asynchronous system with processes prone to crash and Byzantine failures, deterministic algorithms cannot implement consensus [FLP].
  - We use a timing assumption, encapsulated as a **leader detection oracle  $\Omega$** 
    - $\Omega$  periodically designates a trusted leader
    - $\Omega$  is not perfect, may make mistakes
  - Variations of  $\Omega$  can be implemented in partially synchronous systems
    - With crash or Byzantine failures
- 
-



# *Leader-driven consensus*

- Most important paradigm for efficient implementations of consensus
  - Introduced in
    - Viewstamped replication [OL88]
    - Paxos [L96]
    - PBFT [CL02](these formulate it as total-order broadcast)
  - Used in many cloud-serving platforms today
  - Modular presentation of consensus algorithm in 3 steps
- 
-

# Leader-driven consensus



- Leader-driven consensus invokes
  - One instance of Epoch-Change (invokes Omega)
  - Multiple instances of Epoch Consensus
    - Identified by the epoch number and a designated leader

# Preview - Step 1

- Define abstract primitives for
    - Epoch-Change
    - Epoch Consensus
  - Abstractions are valid in both models
  - Leader-driven algorithm for Uniform Consensus (crash faults) and Weak Byzantine Consensus (Byzantine faults)
    - Using Epoch-Change and Epoch Consensus abstractions
- 
-

# *Preview - Step 2*

- Instantiate primitives in model with crash failures
  - According to Viewstamped Replication/Paxos
- Implement Epoch-Change
- Implement Epoch Consensus



# *Preview - Step 3*

- Instantiate primitives in model with Byzantine failures
  - According to PBFT
- Implement Epoch-Change
- Implement Epoch Consensus



# Step 1

**Implement consensus using  
leader-driven algorithm**



# *Eventual Leader Detector*

- Events
    - **Indication**  $\langle \Omega, \text{Trust} \mid p \rangle$ 
      - Indicates that process  $p$  is trusted to be leader
  - Properties
    - **ELD1 (Eventual accuracy)**: Eventually every correct process trusts some correct process.
    - **ELD2 (Eventual agreement)**: Eventually no two correct processes trust a different process.
  - The trusted leader may change over time, different leaders may be elected; only eventually every process follows the same "good" leader.
- 
-

# Epoch-Change (*ec*)

- Events
    - Indication  $\langle ec, \text{StartEpoch} \mid ts, L \rangle$ 
      - Starts epoch  $(ts, L)$ , timestamp  $ts$  and leader  $L$
  - Properties
    - EC1 (Monotonicity): If a correct process starts epoch  $(ts, L)$  and later starts epoch  $(ts', L')$ , then  $ts' > ts$ .
    - EC2 (Consistency): If a correct process starts epoch  $(ts, L)$  and another correct process starts epoch  $(ts, L')$ , then  $L = L'$ .
    - EC3 (Eventual Leadership): Eventually every correct process starts no further epoch; moreover, every correct process starts the same last epoch  $(ts, L)$ , where  $L$  is a correct process.
- 
-



# Epoch Consensus ( $ep$ )

- Associated with timestamp  $ts$  and leader  $L$  (known to all processes)
  - Events
    - Request  $\langle ep, Propose \mid v \rangle$ 
      - Proposes  $v$  for epoch consensus (executed by leader only)
    - Request  $\langle ep, Abort \rangle$ 
      - Aborts this epoch consensus
    - Indication  $\langle ep, Decide \mid v \rangle$ 
      - Outputs decided value  $v$  for epoch consensus
    - Indication  $\langle ep, Aborted \mid s \rangle$ 
      - Signals that this epoch consensus has completed the abort and returns state  $s$
- 
-

# Epoch Consensus (ep)

- Properties
  - **EP1 (Validity)**: If a correct process ep-decides  $v$ , then  $v$  was proposed by the leader of some epoch consensus  $(ts', L)$  with  $ts' \leq ts$ .
  - **EP2 (Uniform Agreement)**: No two [correct\*] processes ep-decide differently.
  - **EP3 (Integrity)**: A correct process ep-decides at most once.
  - **EP4 (Lock-in)**: If a process ep-decides  $v$  in epoch  $ts' < ts$ , no process ep-decides a value different from  $v$ .
  - **EP5 (Termination)**: If the leader  $L$  is correct, has ep-proposed a value and no process aborts, then every correct process eventually ep-decides.
  - (...)

\* for Byzantine epoch consensus

---

---

# *Epoch Consensus (ep)*

- (... Properties)
  - **EP6 (Abort behavior)**: When a correct process aborts, then it eventually completes the abort; plus, a correct process completes an aborts only if it has been aborted before.
- Every process must run a **well-formed sequence** of epoch consensus instances:
  - Only one instance of epoch consensus at a time
  - Associated timestamps monotonically increasing
  - Give state from previous (aborted) instance to next instance

# Leader-driven consensus impl.

Implements  $c^*$  (either  $uc$  or  $wbc$ ), uses  $ec$ ,  $ep$  (multiple instances)

$val := \perp$ ;  $proposed := FALSE$ ;  $decided := FALSE$   
 $(ets, L) := (0, L_0)$ ;  $(newts, newL) := (0, \perp)$

Init. Epoch Consensus inst.  $ep.0$  with timestamp 0 and leader  $L_0$

**upon**  $\langle c^*, \text{Propose } | v \rangle$  **do**  
     $val := v$

**upon**  $\langle ec, \text{StartEpoch } | newts', newL' \rangle$  **do**  
     $(newts, newL) := (newts', newL')$   
    **trigger**  $\langle ep.ets, \text{Abort} \rangle$

**upon**  $\langle ep.ets, \text{Aborted } | s \rangle$  **do**  
     $(ets, L) := (newts, newL)$   
     $proposed := FALSE$   
    Init. Epoch Consensus inst.  $ep.ets$  with timestamp  $ets$ , leader  $L$ ,  
    and state  $s$

---

---

# *Leader-driven consensus impl.*

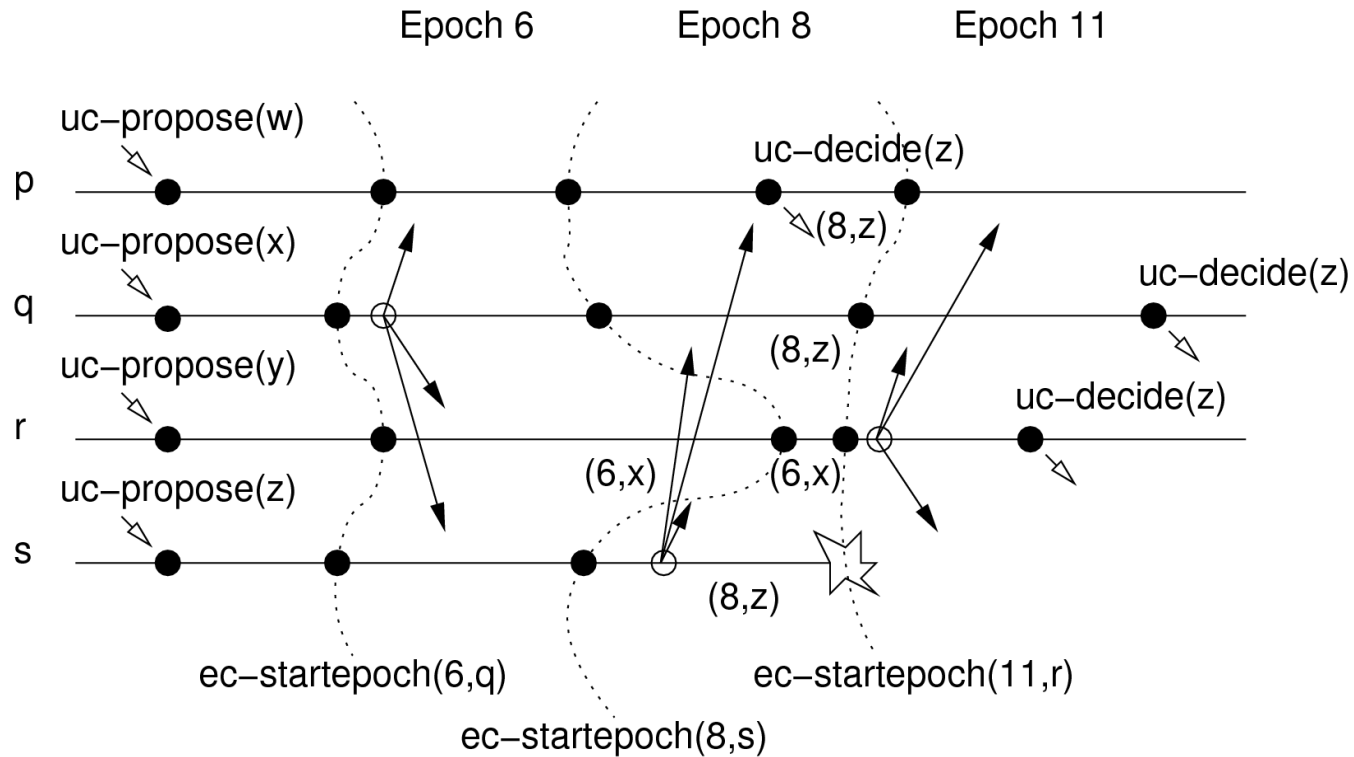
(...)

```
upon L = self  $\wedge$  val  $\neq$   $\perp$   $\wedge$   $\neg$ proposed do  
  proposed := TRUE  
  trigger <ep.ets, Propose | val>
```

```
upon <ep.ets, Decide | v> do  
  if  $\neg$ decided then  
    decided := TRUE  
    trigger <c*, Decide | v>
```



**Ex.**



- Every process (p, q, r, s) uc-proposes a different value
- Epoch 6 has leader q
  - q ep-proposes(x); only r receives it
  - Then epoch is aborted
- Epoch 8 has leader s
  - s ep-proposes(z), processes p, q, s receive it
  - p ep-decides(z); then s crashes and epoch is aborted
- Epoch 11 has leader r
  - r ep-proposes(y); processes q and r ep-decide(z)

# Correctness

- **Termination (UC1 / WBC1)**
    - From EC3 (eventual leadership), EP5 (termination) and algorithm
  - **Validity (UC2) / Weak Validity (WBC2)**
    - From EP1 (validity) and algorithm
  - **Integrity (UC3)**
    - Immediate from algorithm
  - **Uniform Agreement (UC4 / WBC4)**
    - From algorithm and EP2 (agreement) and EP4 (lock-in)
- 
-

## **Step 2**

**Implement epoch-change and  
epoch consensus in crash-  
failure model**





# *Implementing epoch-change*

- Use eventual leader detector ( $\Omega$ )
- Maintain current trusted leader and timestamp
- When  $\Omega$  indicates a different leader is trusted
  - Increment timestamp
  - Broadcast a NEWEPOCH message (with leader and timestamp)
- When delivering a NEWEPOCH message
  - Trigger start of new epoch

(Only a sketch; details omitted)

---

---

# *Implementing epoch consensus*

- Read/write epoch consensus algorithm
    - Analogous to replicated implementation of a shared single-writer register
  - State consists of a timestamp/value pair
  - Leader reads state and looks for a value
    - Chooses value with highest timestamp
    - If no value found, takes value from its ep-proposal
    - Writes the chosen value
  - Decide once a quorum of processes ( $> N/2$ ) accept the written value
- 
-

# *Read/write epoch consensus*

Implements **ep**, uses **pl**, **beb** ( $N > 2f$ ), with ts. **ets** and leader **L**

**upon**  $\langle \mathbf{ep}, \text{Init} \mid (\text{valts}, \text{val}) \rangle$  **do**  
     $\text{tmpval} := \perp$ ;  $\text{states} := [\perp]^N$ ;  $\text{accepted} := 0$

**upon**  $\langle \mathbf{ep}, \text{Propose} \mid v \rangle$  **do**  
     $\text{tmpval} := v$   
    **trigger**  $\langle \mathbf{beb}, \text{Broadcast} \mid [\text{READ}] \rangle$

**upon**  $\langle \mathbf{beb}, \text{Deliver} \mid L, [\text{READ}] \rangle$  **do**  
    **trigger**  $\langle \mathbf{pl}, \text{Send} \mid L, [\text{STATE}, \text{valts}, \text{val}] \rangle$

**upon**  $\langle \mathbf{pl}, \text{Deliver} \mid q, [\text{STATE}, \text{ts}, v] \rangle$  **do**  
     $\text{states}[q] := (\text{ts}, v)$

**upon**  $\#(\text{states}) > N/2$  **do**  
     $(\text{ts}, v) := \text{highest}(\text{states})$ ;  $\text{states} := [\perp]^N$   
    **if**  $v \neq \perp$  **then**  $\text{tmpval} := v$   
    **trigger**  $\langle \mathbf{beb}, \text{Broadcast} \mid [\text{WRITE}, \text{tmpval}] \rangle$

---

---

# *Read/write epoch consensus*

(...)

**upon** <beb, Deliver | L, [WRITE, v]> **do**  
 (valts,val) := (ets,v)  
 **trigger** <pl, Send | L, [ACCEPT]>

**upon** <pl, Deliver | q, [ACCEPT]> **do**  
 accepted := accepted + 1

**upon** accepted > N/2 **do**  
 accepted := 0  
 **trigger** <beb, Broadcast | [DECIDED, tmpval]>

**upon** <pl, Deliver | L, [DECIDED, v]> **do**  
 **trigger** <ep, Decide | v>

**upon** <ep, Abort> **do**  
 **trigger** <ep, Aborted | (valts,val)>

---

---

# Correctness (1)

- **Validity (EP1)**
    - The ep-decided value was written by **L**
    - If any STATE msg. contains a value, **L** writes this
      - This value has been written by some leader
    - Otherwise, **L** writes its own ep-proposed value
  - **Uniform Agreement (EP2)**
    - Immediate from DECIDED msg. in algorithm
  - **Integrity (EP3)**
    - Immediate from algorithm
  - **Lock-in (EP4)**
    - A write-quorum ( $> N/2$ ) stored **v** before sending the ACCEPT msg. in previous epoch  $ts' < ts$
    - Processes passed it in state to subsequent epochs
    - Then, **L** reads **v** from at least one STATE msg. in read-quorum ( $> N/2$ )
- 
-

# *Correctness (2)*

- **Termination (EP5)**
  - If leader **L** is correct, then every process eventually decides
- **Abort behavior (EP6)**
  - Immediate from algorithm



## **Step 3**

**Implement epoch-change and epoch consensus in Byzantine-failure model**



# Implementing Byzantine epoch-change

- Use Byzantine eventual leader detector (**bld**)
  - **bld** allows application to complain when no progress
- Maintain current trusted leader and timestamp
- When **bld** indicates a different leader is trusted
  - Increment timestamp
  - Derive leader from timestamp (deterministically)
  - Broadcast a NEWEPOCH message (with timestamp)
- When delivering  $> f$  NEWEPOCH messages
  - Trigger start of new epoch

(Only a sketch; details omitted)

---



# *Implementing Byzantine epoch consensus (1)*

- Byzantine read/write epoch consensus alg.
  - Analogous to replicated implementation of a Byz. shared single-writer register
- State consists of timestamp/value pair and set of "previously" written values
- Leader should read state of all processes and determine value to write
  - But cannot trust single leader
  - Thus, all processes read state and determine value
    - Encapsulated by a **conditional collect** primitive

(...)

---

---

# Implementing Byzantine epoch consensus (2)

- Processes choose value with highest timestamp
    - If no value found, only then leader is free to take the value from its ep-proposal
  - All processes write the chosen value
    - Broadcast WRITE message to all
  - When receiving WRITE msg. with value  $v$  from  $> (N+f)/2$  processes, then store  $v$ 
    - Broadcast ACCEPT msg. message to all
  - When receiving ACCEPT msg. with  $v$  from  $> (N+f)/2$  processes, then ep-decide
- 
-

# Conditional Collect (cc)

- Parameterized by a predicate **C** and leader **L**
    - Leader **L** will also be the leader of the epoch
  - Events
    - Request **<cc, Input | m>**
      - Inputs message **m**
    - Indication **<cc, Collected | M>**
      - Outputs vector **M** of collected messages or **UNDEFINED**
  - Properties
    - **CC1 (Consistency)**: If **L** is correct, every correct pr. collects the same **M**, which contains at least **N-f** messages different from **UNDEFINED**.
    - **CC2 (Integrity)**: If a correct pr. collects **M** with **M[p] ≠ UNDEFINED** and **p** is correct, then **p** has input **m**.
    - (...)
- 
-

# Conditional Collect (cc)

- (... Properties)
  - **CC3 (Termination)**: If **L** is correct and all correct pr. input messages such that they satisfy **C**, then every correct process eventually collects **M** s.t. **C(M)**.
- Note
  - Every process inputs a message
  - Output is vector of such messages, one per process
  - If **L** correct, then output **M** satisfies the predicate
    - Otherwise, may not terminate

# Byz. read/write epoch cons. (1)

Implements **ep**, uses **al**, **abeb**, **cc** ( $N > 3f$ ), with ts. **ets** and leader **L**

**upon**  $\langle \mathbf{ep}, \text{Init} \mid (\text{valts}, \text{val}, \text{ws}) \rangle$  **do**  
    written :=  $[\perp]^N$ ; accepted :=  $[\perp]^N$

**upon**  $\langle \mathbf{ep}, \text{Propose} \mid v \rangle$  **do**  
    **if** val =  $\perp$  **then** val := v  
    **trigger**  $\langle \mathbf{abeb}, \text{Broadcast} \mid [\text{READ}] \rangle$

**upon**  $\langle \mathbf{abeb}, \text{Deliver} \mid L, [\text{READ}] \rangle$  **do**  
    **trigger**  $\langle \mathbf{cc}, \text{Input} \mid [\text{STATE}, \text{valts}, \text{val}, \text{ws}] \rangle$

**upon**  $\langle \mathbf{cc}, \text{Collected} \mid S \rangle$  **do**  
    // note, for all p :  $S[p] = [\text{STATE}, \text{ts}, v, \text{ws}]$  or UNDEFINED  
    tmpval :=  $\perp$   
    **if**  $\exists \text{ts} \geq 0, v \neq \perp$  from S : **binds**(ts,v,S) **then** tmpval := v  
    **else if**  $\exists v \neq \perp : \mathbf{unbound}(S) \wedge v \in S[L]$  **then** tmpval := v  
    **if** tmpval =  $\perp$  **then** halt  
    (...)

---

---

# Byz. read/write epoch cons. (2)

(... **upon** <cc, Collected | S> **do**  
  **if**  $\exists ts : (ts, tmpval) \in ws$  **then**  $ws := ws \setminus \{(ts, tmpval)\}$   
   $ws := ws \cup \{(ets, tmpval)\}$   
  **trigger** <abeb, Broadcast | [WRITE, tmpval]>

**upon** <abeb, Deliver | p, [WRITE, v]> **do**  
  written[p] := v  
  **if**  $\exists v : \#\{p | \text{written}[p]=v\} > (N+f)/2$  **then**  
    (vals, val) := (ets, v)  
    written :=  $[\perp]^N$   
    **trigger** <abeb, Broadcast | [ACCEPT, val]>

**upon** <abeb, Deliver | q, [ACCEPT, v]> **do**  
  accepted[p] := v  
  **if**  $\exists v : \#\{p | \text{accepted}[p]=v\} > (N+f)/2$  **then**  
    written :=  $[\perp]^N$   
    **trigger** <ep, Decide | v>

---

---

# Byz. read/write epoch cons. (3)

- Predicate **binds**( $ts, v, S$ ):
    - Whether **( $ts, v$ )** is confirmed by  **$> (N+f)/2$**  entries in  **$S$**  to be value associated to highest timestamp, and
    - Value  **$v$**  has not been invented out of thin air
      - Hence, processes write this value again
  - Predicate **unbound**( $S$ ):
    - Evidence that no value can be bound by  **$S$** 
      - Hence, processes write value of the leader
  - Predicate **sound**( $S$ ) for **cc**:
    - **$\exists(ts, v)$**  such that **binds**( $ts, v, S$ )  $\vee$  **unbound**( $S$ )
- 
-

# Correctness (1)

- **Validity (EP1)**
    - The ep-decided value  $v$  was written in the epoch
    - Either collected vector  $S$  satisfies  $\text{bound}(ts, v, S)$ 
      - Then  $v$  has been written in an "earlier" epoch
    - Otherwise, take ep-proposed value of  $L$
  - **Uniform Agreement (EP2)**
    - Immediate from quorum of ACCEPT msgs.
  - **Integrity (EP3)**
    - Immediate from algorithm
  - **Lock-in (EP4)**
    - A write-quorum ( $> (N+f)/2$ ) stored  $v$  before sending an ACCEPT msg. in previous epoch  $ts' < ts$
    - Processes passed it in state to subsequent epochs
    - Then, conditional collect determines from STATE msgs. in a quorum ( $> (N+f)/2$ ) that such  $v$  exists
- 
-



# Correctness (2)

- **Termination (EP5)**
  - If leader **L** is correct, then every process eventually decides
    - Given termination of conditional collect (CC3)
    - Same as termination of Byz. reliable broadcast
- **Abort behavior (EP6)**
  - Immediate from algorithm (omitted)



# *Summary*

- Same leader-driven consensus algorithm with crash failures and Byzantine failures
    - Using abstract primitives of epoch-change and epoch consensus
  - Primitives implemented in crash model
    - Paxos consensus algorithm
  - Primitives implemented in Byzantine model
    - PBFT consensus algorithm
- 
-

# Coda



# *Summary*

- Distributed programming defines abstractions of
    - Reliable broadcast
    - Shared memory
    - Consensus
  - Implementations in distributed systems
  - By group of processes, which are subject to
    - Crash failures
    - Attacks/Byzantine failures
- 
-

**For everything else, see the  
book.**

**[www.distributedprogramming.net](http://www.distributedprogramming.net)**

