Asymmetric Trust in Distributed Systems

Luca Zanolini
University of Bern

Supervisor: Prof. Dr. **Christian Cachin**

6th July 2023

Secure distributed systems rely on **trust**

UNIVERSITÄT BERN

- Specifies the **failures** that a system can tolerate.
- Determines the **conditions** under which a system operates correctly.
- Defined through a **fail-prone system**.
- **Fail-prone systems** are useful tools for the design of distributed algorithms.

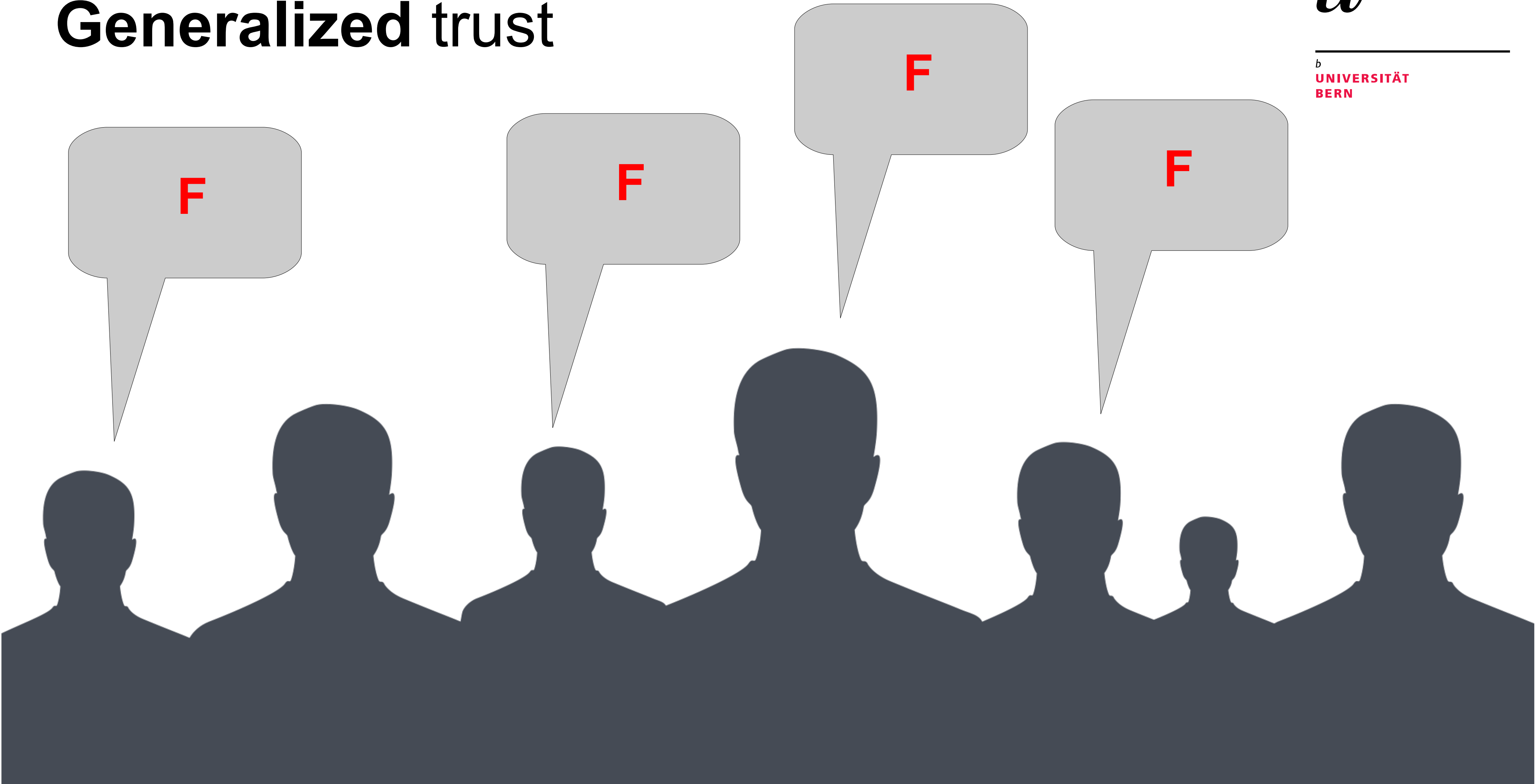
Permissioned systems

UNIVERSITÄT BERN

- **P** = $\{p_1, \ldots, p_n\}$.
- Full system membership is public knowledge.
- Trust assumptions are public knowledge.
- Participants do not lie about their trust assumptions.

Generalized trust

b UNIVERSITÄT BERN



Byzantine quorum systems

UNIVERSITÄT BERN

- Set of processes $\mathbf{P} = \{p_1, \ldots, p_n\}$.

- Fail-prone system $\mathbf{F} \subseteq 2^{\mathbf{P}}$: all processes in some $F \in \mathbf{F}$ may fail together.

- Quorum system $\mathbf{Q} \subseteq 2^{\mathbf{P}}$, where any $Q \in \mathbf{Q}$ is a **quorum**, if and only if:

  – **Consistency**:

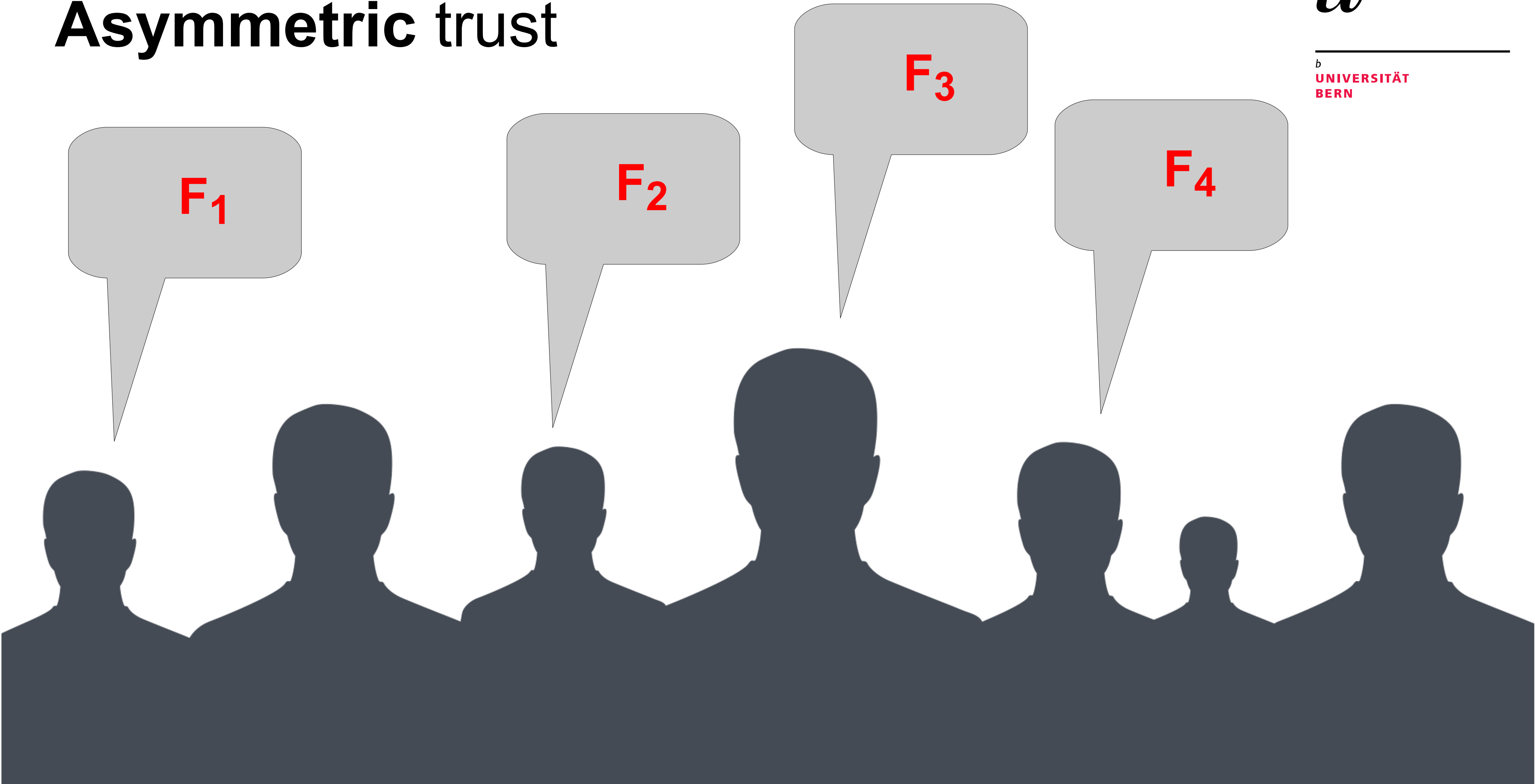
$$\forall\ Q_1, Q_2 \in \mathbf{Q}, \forall F \in \mathbf{F} : Q_1 \cap Q_2 \not\subseteq F.$$

  – **Availability**:

$$\forall F \in \mathbf{F} : \exists Q \in \mathbf{Q} : F \cap Q = \emptyset.$$

[Malkhi & Reiter, 1998]

Asymmetric trust



Asymmetric Byzantine quorum systems

- Set of processes $\mathbf{P} = \{p_1, \ldots, p_n\}$.
- Fail-prone systems $\mathbf{F_i} \subseteq 2^{\mathbf{P}}$ for p_i.
- Quorum systems $\mathbf{Q_i} \subseteq 2^{\mathbf{P}}$, where any $Q_i \in \mathbf{Q_i}$ is a **quorum** for p_i ,if and only if:
  - **Consistency**:

$$\forall\ Q_i \in \mathbf{Q_i},\ \forall\ Q_j \in \mathbf{Q_j},\ \forall\ F_{ij} \in \mathbf{F_i}^* \cap \mathbf{F_j}^*:\ Q_i \cap Q_j \not\subseteq F_{ij}.$$

  - **Availability**:

$$\forall\ F_i \in \mathbf{F_i} : \exists\ Q_i \in \mathbf{Q_i} : F_i \cap Q_i = \emptyset.$$

[Cachin & Tackmann, 2019]

In the **asymmetric** trust model

- **Faulty**: A process $p_i \in F$ is called *faulty*
- **Naive**: A correct process p_i for which $F \notin \mathbf{F_i}^*$ is called *naive*
- **Wise**: A correct process for which $F \in \mathbf{F_i}^*$ is called *wise*

Guild

b UNIVERSITÄT BERN

Some of our results

- Better understanding of the **relationship** between wise and naive processes.

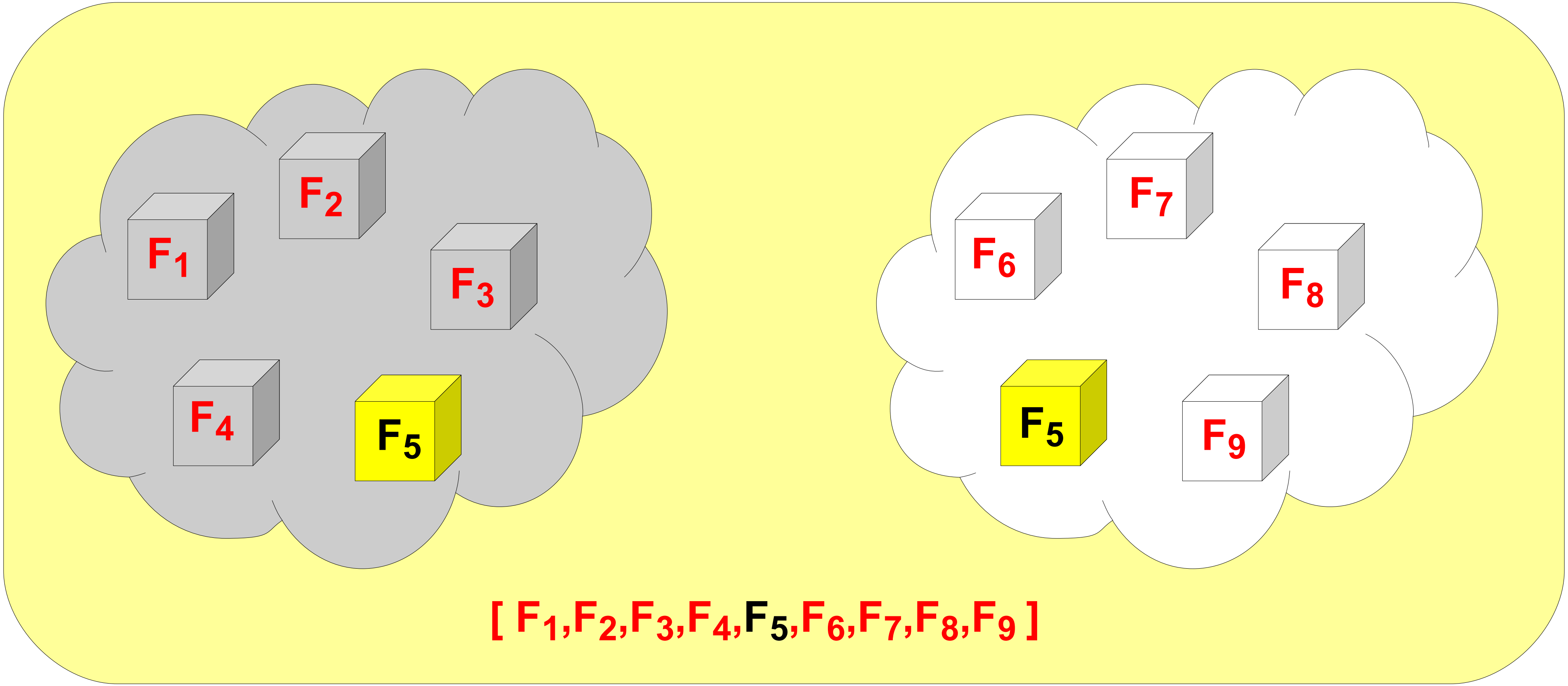
- **Uniqueness** of the guild in an execution.

- Importance of a guild in **kernel-based protocols**, e.g., Bracha broadcast.

- Tolerated system **T** $=$ $\{$**P** \ **G**, for any possible guild **G** $\}$

- **Composition rule** among asymmetric-trust based systems.

Find a (deterministic) **composition** rule



First asynchronous Byzantine **consensus protocol** with asymmetric trust

UNIVERSITÄT BERN

- It uses **randomization**
- Signature-free
- Round-based
- Suitable for applications in **blockchain** networks
- Builds on the protocol by Mostéfaoui et al. (PODC 2014)

Signature-Free Asynchronous Byzantine Consensus with $t < n/3$ and $O(n^2)$ Messages

Achour Mostéfaoui
LINA, Université de Nantes
44322 Nantes, France
Achour.Mostefaoui@univ-nantes.fr

Hamouma Moumen
University of Bejaia
Bejaia, Algeria
moumenh@gmail.com

Michel Raynal
Institut Universitaire de France
& IRISA, Université de Rennes
raynal@irisa.fr

The (original) protocol

i. Binary validated **broadcast**
ii. Randomized **consensus**
   – *Uses a **common coin***

bv-broadcast(b)

The (original) protocol

i. Binary validated **broadcast**

ii. Randomized **consensus**

– *Uses a* ***common coin***

bv-broadcast(b) $\rightarrow$ *bv-deliver(b)*

2f+1

The (original) protocol

i. Binary validated **broadcast**

ii. Randomized **consensus**

– *Uses a **common coin***

bv-broadcast(b) → bv-deliver(b) → [AUX,b] to all

2f+1

The (original) protocol

i. Binary validated **broadcast**

ii. Randomized **consensus**

– *Uses a* ***common coin***

bv-broadcast(b) → bv-deliver(b) → [AUX,b] to all → b received

2f+1 *2f+1*

The (original) protocol

i. Binary validated **broadcast**

ii. Randomized **consensus**

– *Uses a* ***common coin***

bv-broadcast(b) → bv-deliver(b) → [AUX,b] to all → b received → release-coin

2f+1 2f+1

The (original) protocol

i. Binary validated **broadcast**

ii. Randomized **consensus**

– *Uses a **common coin***

bv-broadcast(b) → bv-deliver(b) → [AUX,b] to all → b received → release-coin → output-coin(s)

2f+1 *2f+1*

The (original) protocol

i. Binary validated **broadcast**

ii. Randomized **consensus**

– *Uses a **common coin***

bv-broadcast(b) → bv-deliver(b) → [AUX,b] to all → b received → release-coin → output-coin(s) → if b = s, rbc-decide(b)

2f+1 *2f+1*

if b ≠ s, bv-broadcast(b)

if {0,1}, bv-broadcast(s)

B

Liveness issue!

The network **reorders** messages between correct processes and delays them until the coin value becomes known.

Fixing the problem

i. **FIFO** ordering on the reliable point-to-point links, including the messages exchanged by the coin implementation

- the adversary may no longer exploit its knowledge of the coin value to prevent termination.

ii. Allow the set B to **dynamically change** while the coin protocol executes.

iii. Our protocol **does not** execute rounds forever, as in the original formulation.

The (asymmetric) protocol

b UNIVERSITÄT BERN

i. Asymmetric binary validated **broadcast**

ii. Asymmetric randomized **consensus**

– *Uses an* ***asymmetric common coin***

abv-broadcast(b) → abv-deliver(b) → [AUX,b] to all → b received → release-coin → output-coin(s) → if b = s, arbc-decide(b)

Q_i Q_i

Asymmetric strong Byzantine consensus

b UNIVERSITÄT BERN

In all executions with a guild:

- [**Probabilistic termination**] Every wise process decides with probability 1.
- [**Strong validity**] A wise process only decides a value that has been proposed by some processes in the maximal guild.
- [**Integrity**] No correct process decides twice.
- [**Agreement**] No two wise processes decide differently.

Permissionless systems

- **P** = {p_1, p_2 ,...}.
- Knowledge of the full system membership is **not** available.
- Trust assumptions are (**partially**) public knowledge.
- Participants **can lie** about their trust assumptions.

Our model

UNIVERSITÄT BERN

- Each process p_i makes assumptions about a set $P_i \subseteq$ **P** $= \{p_1, p_2, \ldots\}$ called p_i's trusted set, using a fail-prone system $\mathbf{F_i}$ over P_i .

- Point-to-point communication & best-effort gossip primitive.

- Each process p_i continuously discovers new processes and learns their assumptions.

- A **permissionless fail-prone system** is an array:

$$\mathbb{F} = [(P_1, \mathbf{F_1}), (P_2, \mathbf{F_2}), \ldots,]$$

Assumptions of p_i

We say that the assumptions of a process p_i **are satisfied** in an execution if the set A of processes that fail is such that there exists a fail-prone set $F \in \mathbf{F_i}$ such that:

i. $A \cap P_i \subseteq F$;

ii. the assumptions of every member of $P_i \setminus F$ **are satisfied**.

Assumptions of p_i

We say that the assumptions of a process p_i **are satisfied** in an execution if the set A of processes that fail is such that there exists a fail-prone set $F \in \mathbf{F_i}$ such that:

i. $A \cap P_i \subseteq F$;

ii. the assumptions of every member of $P_i \setminus F$ **are satisfied**.

If p_i **has its assumptions satisfied** in an execution, we say that p_i **tolerates the execution**.

A set of processes L **tolerates a set of processes** A if and only if every process p_i in $L \setminus A$ tolerates an execution with set of faulty processes A.

A **new kind** of failure assumptions

A participant's assumption are not only about failures, but *also about whether other participants make correct assumptions*.

A **new kind** of failure assumptions

A participant's assumption are not only about failures, but *also about whether other participants make correct assumptions*.

How do we define **quorums**?

- Global **intersection** property among quorums?
- Malicious processes can **lie** about their assumptions.

Views

A **view** $\mathbb{V} = [\mathbf{V_1},\mathbf{V_2}, \ldots]$ is an array with one entry $\mathbb{V}[j] = \mathbf{V_j}$ for each process p_i such that:

i. either $\mathbf{V_j}$ is the special value $\perp$; or
ii. $\mathbf{V_j} = (P_j , \mathbf{F_j})$ consists of a set of processes P_j and a fail-prone system $\mathbf{F_j}$.

A process p_i 's view is what p_i thinks other's assumptions are. However, such view might contain lies from Byzantine processes.

Views

A **view** $\mathbb{V} = [\mathbf{V_1},\mathbf{V_2}, \dots]$ is an array with one entry $\mathbb{V}[j] = \mathbf{V_j}$ for each process p_i such that:

i. either $\mathbf{V_j}$ is the special value $\perp$; or
ii. $\mathbf{V_j} = (P_j, \mathbf{F_j})$ consists of a set of processes P_j and a fail-prone system $\mathbf{F_j}$.

A process p_i 's view is what p_i thinks other's assumptions are. However, such view might contain lies from Byzantine processes.

Given a set of faulty processes A in an execution, we say that a view $\mathbb{V}$ is A-**resilient** if and only if for every process $p_i \notin A$, either $\mathbb{V}[i] = \perp$ or $\mathbb{V}[i] = \mathbb{F}[i]$.

Quorum **function**

A quorum is a set of processes that satisfies the assumptions of every one of its members.

Quorum **function**

A quorum is a set of processes that satisfies the assumptions of every one of its members.

The **quorum function** $\mathcal{Q}$: **P** x $\mathcal{V}$ -> $2^{\mathbf{P}}$ maps a process p_i and a view $\mathbb{V}$ to a set of processes such that $Q \in \mathcal{Q}(p_i, \mathbb{V})$ if and only if:

i. there exists $F \in \mathbf{F_i}$ for p_i such that $P_i \setminus F \subseteq Q$;
ii. for every process $p_j \neq p_i \in Q$ with $\mathbb{V}[i] \neq \perp$ and $\mathbf{V_j} = (P_j, \mathbf{F_j})$, there exists $F \in \mathbf{F_j}$ for p_j such that $P_j \setminus F \subseteq Q$.

Every element of $Q \in \mathcal{Q}(p_i, \mathbb{V})$ is called a **permissionless quorum** for p_i.

Leagues

A league is a set of processes that enjoys quorum intersection and quorum availability in all executions that it tolerates.

Leagues

UNIVERSITÄT BERN

A league is a set of processes that enjoys quorum intersection and quorum availability in all executions that it tolerates.

A set of processes L is a **league** for the quorum function $\mathcal{Q}$ if and only if:

i. **Consistency**: for every set $A \subseteq P$ **tolerated by** L, for every two A-resilient **views** $\mathbb{V}$ and $\mathbb{V}'$, for every two **processes** p_i and $p_j \in L \setminus A$, and for every two **quorums** $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_i, \mathbb{V}')$ it holds $(Q_i \cap Q_j) \setminus A \neq \emptyset$;

ii. **Availability**: for every set $A \subseteq P$ tolerated by L and for every **process** $p_i \in L \setminus A$, there exists a **quorum** $Q_i \in \mathcal{Q}(p_i, \mathbb{F})$ such that $Q_i \subseteq L \setminus A$.

Permissionless Byz. reliable broadcast

UNIVERSITÄT BERN

For every league L and every execution tolerated by L:

- [**Validity**] If a correct process p_s broadcasts a value v, the all correct processes in L eventually deliver v.

- [**Integrity**] For any value v, every correct process delivers v at most once. Moreover, if the sender p_s is correct and the receiver is correct and in L, then v was previously broadcast by p_s.

- [**Consistency**] If a correct process in L delivers some value v and another correct process in L delivers some value v', then v = v'.

- [**Totality**] If a correct process in L delivers some value v, then all correct processes in L eventually deliver some value.

Open questions

UNIVERSITÄT BERN

- Asymmetric threshold cryptography.
- Asymmetric leader-based consensus protocols.
- More composition rules.
- Byzantine consensus protocols in the permissionless setting.

Thank **you**!

Bibliography

- O. Alpos, C. Cachin, and **L. Zanolini**, “How to trust strangers: Composition of byzantine quorum systems,” in **SRDS** 2021
- C. Cachin, G. Losa, and **L. Zanolini**, “Quorum systems in per- missionless networks,” in **OPODIS** 2022
- C. Cachin and **L. Zanolini**, “Asymmetric asynchronous byzantine consensus,” in **CBT** 2021 - **ESORICS** 2021
- C. Cachin and **L. Zanolini**, “Brief announcement: Revisiting signature-free asynchronous byzantine consensus,” in **DISC** 2021

Asymmetric common coin

A protocol for asymmetric common coin satisfies the following properties:

- [**Termination**] In all the executions with a guild, every process in the maximal guild eventually outputs a coin value.
- [**Unpredictability**] In all the executions with a guild, no process has any information about the value of the coin before at least a kernel for all wise processes, which consists of correct processes, has released the coin.
- [**Matching**] In all the executions with a guild, with probability 1 every process in the maximal guild outputs the same coin value.
- [**No bias**] The distribution of the coin is uniform over {0,1}.

Algorithm 2 Asymmetric common coin for round *round* (code for p_i)

State

$\mathcal{H}$: set of all possible guilds

$share[\mathcal{G}][j]$: if $p_i \in \mathcal{G}$, this holds the share received from p_j for guild $\mathcal{G}$; initially $\perp$

upon event *release-coin* **do**

  for all $\mathcal{G} \in \mathcal{H}$ **such that** $p_i \in \mathcal{G}$ **do**

    let $s_{i\mathcal{G}}$ be the share of p_i for guild $\mathcal{G}$

    for all $p_j \in \mathcal{P}$ **do**

      send message [SHARE, $s_{i\mathcal{G}}$, $\mathcal{G}$, *round*] to p_j

upon receiving a message [SHARE, s, $\mathcal{G}$, r] from p_j **such that** $r = round$ **and** $p_j \in \mathcal{G}$ **do**

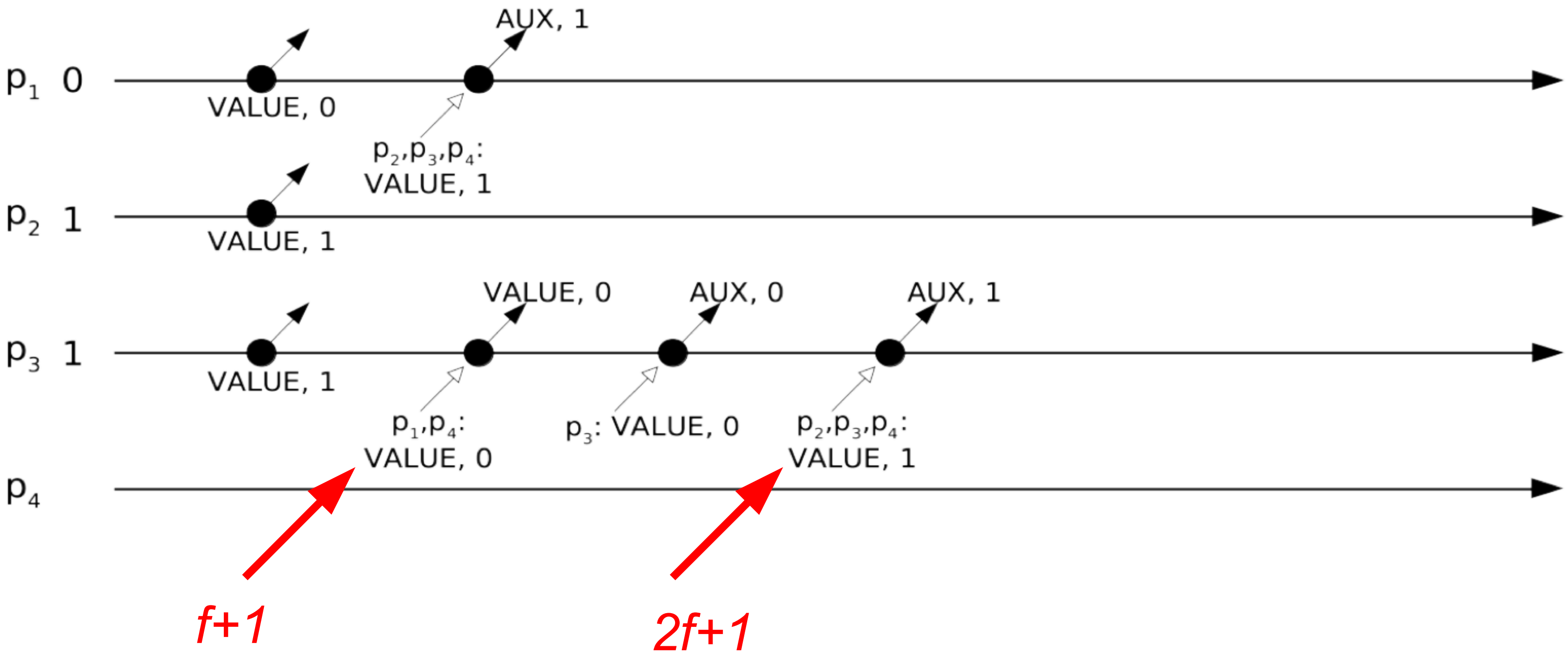
  if $share[\mathcal{G}][j] = \perp$ **then**

    $share[\mathcal{G}][j] \leftarrow s$

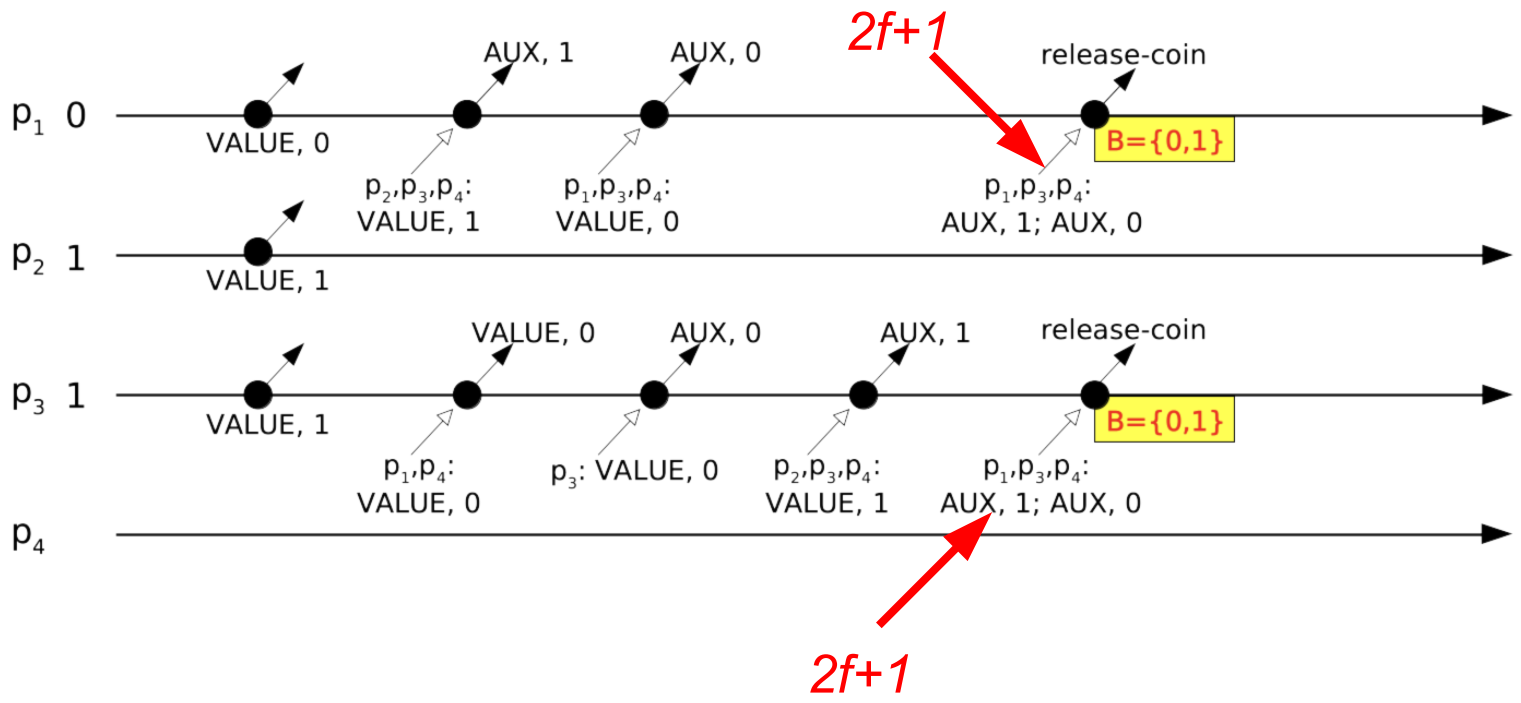
upon exists $\mathcal{G}$ **such that for all** j with $p_j \in \mathcal{G}$, it holds $share[\mathcal{G}][j] \neq \perp$ **do**

  $s \leftarrow \sum_{j:p_j \in \mathcal{G}} share[\mathcal{G}][j]$

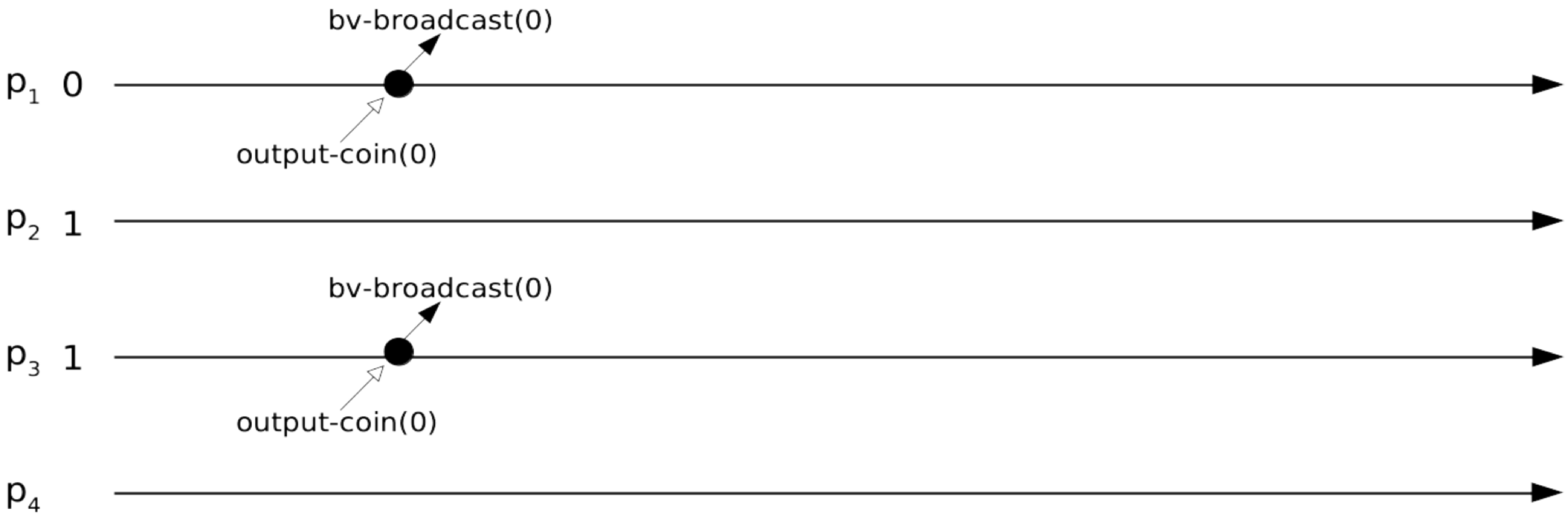
  output *output-coin*(s)



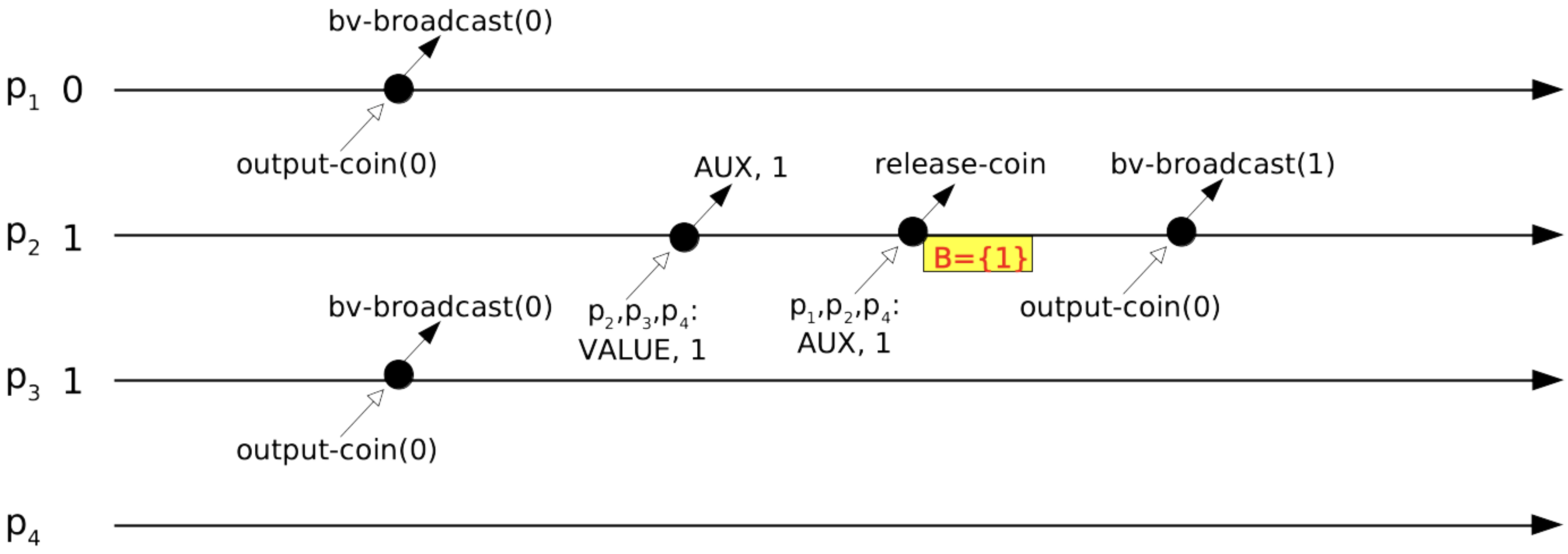
AUX, 1
p_1 0
VALUE, 0
p_2,p_3,p_4:
VALUE, 1
p_2 1
VALUE, 1
VALUE, 0
AUX, 0
AUX, 1
p_3 1
VALUE, 1
p_1,p_4:
VALUE, 0
p_3: VALUE, 0
p_2,p_3,p_4:
VALUE, 1
p_4
f+1
2f+1



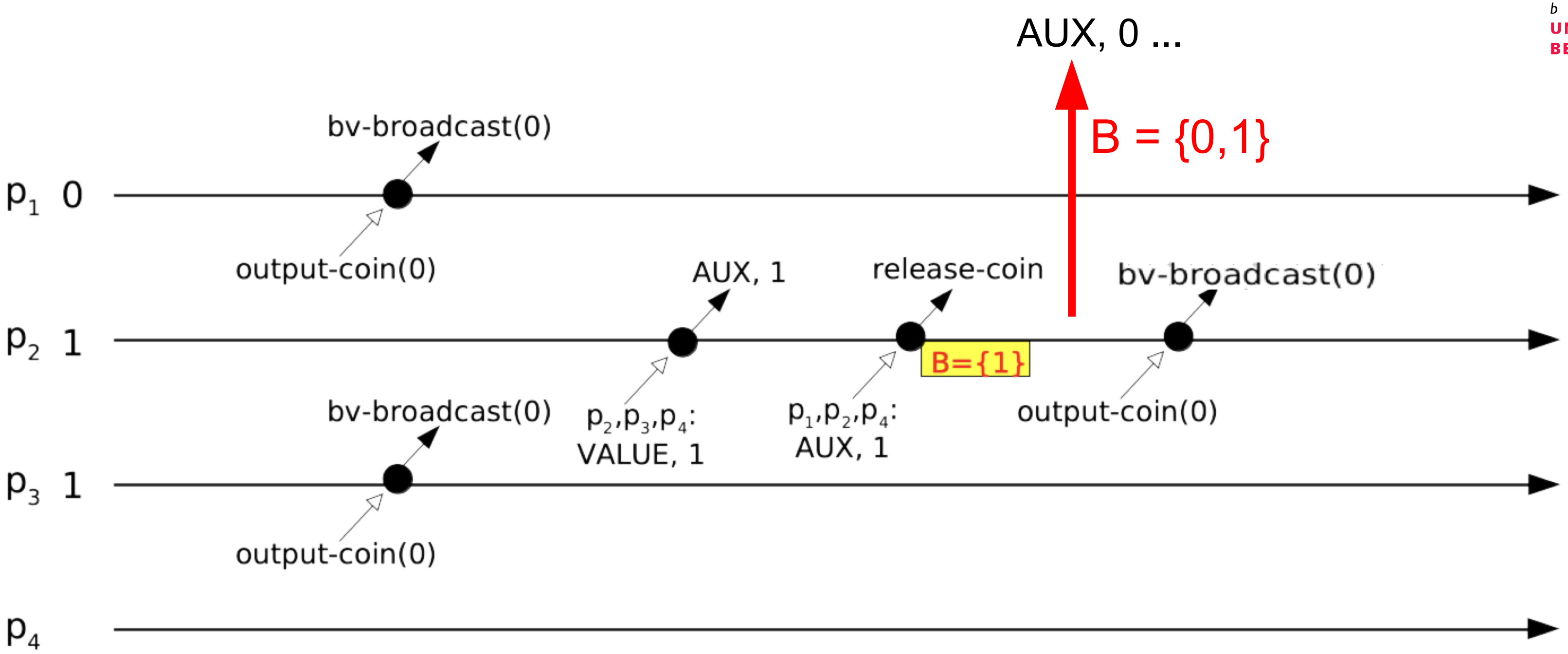
2f+1
AUX, 1
AUX, 0
release-coin
p1 0
VALUE, 0
B={0,1}
p2,p3,p4:
VALUE, 1
p1,p3,p4:
VALUE, 0
p1,p3,p4:
AUX, 1; AUX, 0
p2 1
VALUE, 1
VALUE, 0
AUX, 0
AUX, 1
release-coin
p3 1
VALUE, 1
B={0,1}
p1,p4:
VALUE, 0
p3: VALUE, 0
p2,p3,p4:
VALUE, 1
p1,p3,p4:
AUX, 1; AUX, 0
p4
2f+1



Random coin = 0



Random coin = 0


AUX, 0 ...
B = {0,1}
bv-broadcast(0)
p1 0
output-coin(0)
AUX, 1
release-coin
bv-broadcast(0)
p2 1
B={1}
bv-broadcast(0)
p2,p3,p4:
VALUE, 1
p1,p2,p4:
AUX, 1
output-coin(0)
p3 1
output-coin(0)
p4