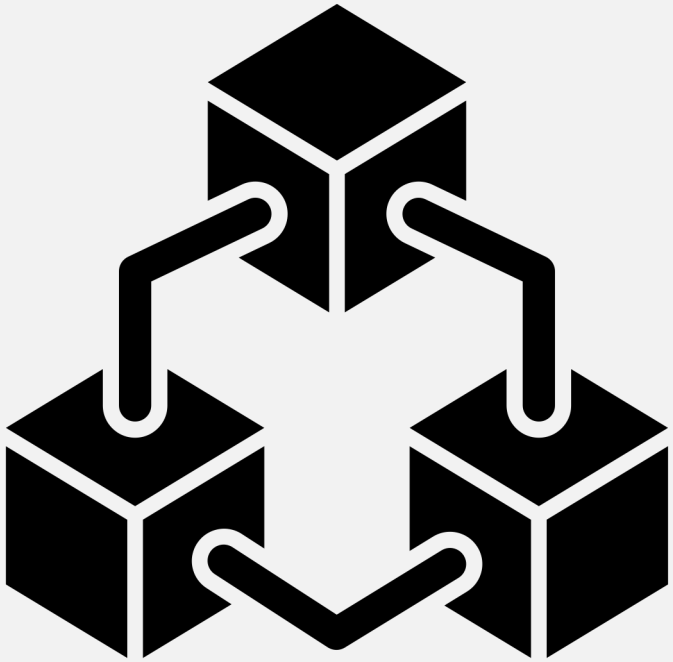
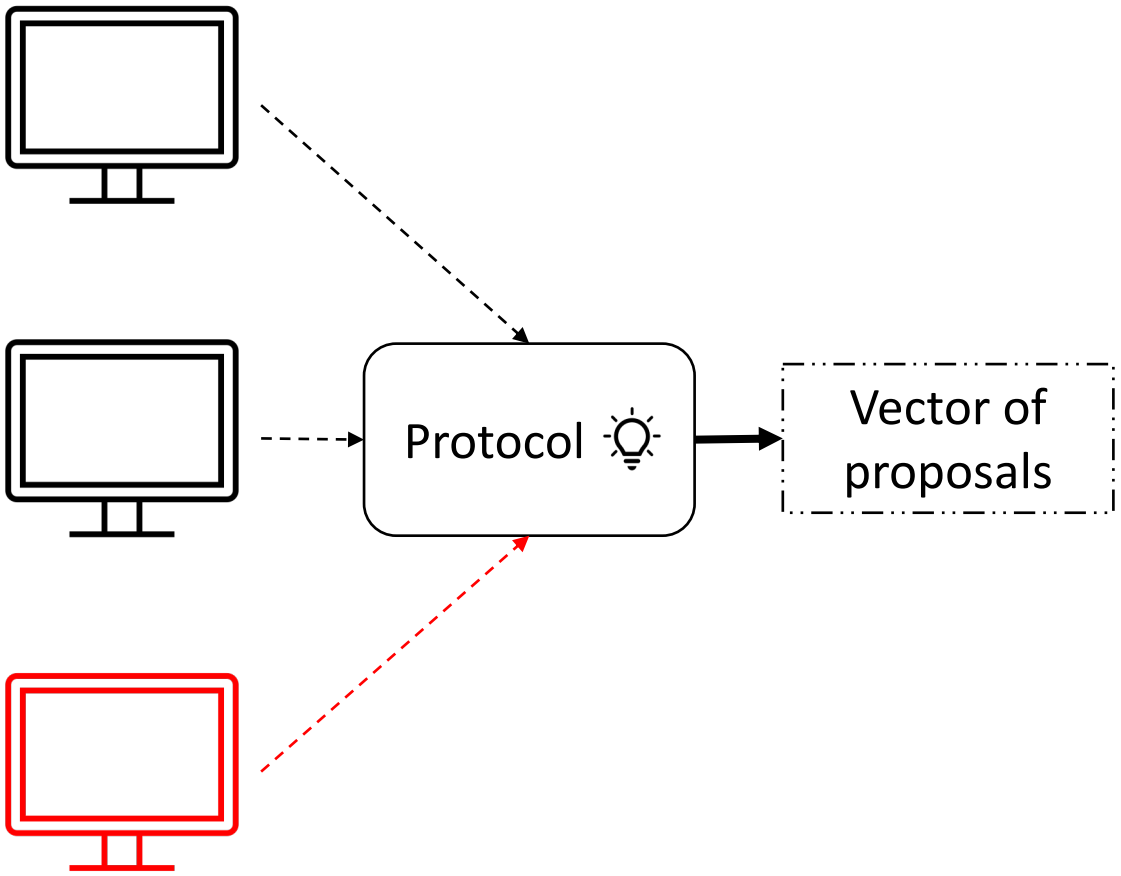


Improving the Complexity of Byzantine Vector Consensus

Manos Chatzakis
emmanouil.chatzakis@epfl.ch

EPFL

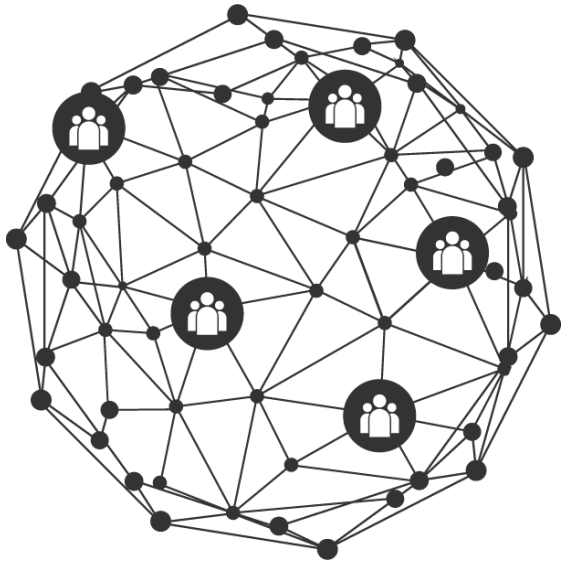
What is Byzantine Vector Consensus?

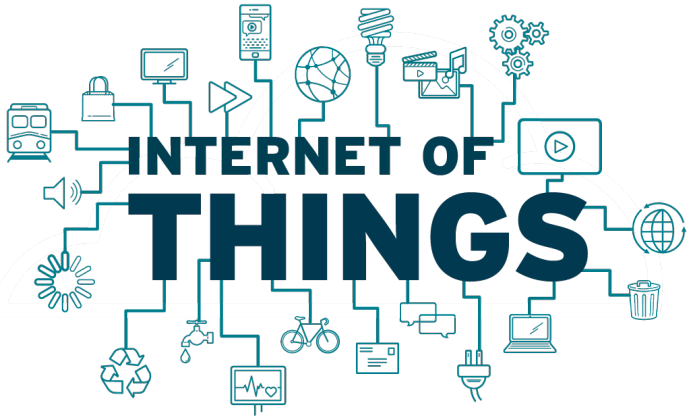


- System: n processes
  - f Byzantine
  - $n = 3f + 1$
- Decide a **vector of $(n - f)$ distinct proposals**
- Partial Synchrony: Delays are initially unbounded, until $\boldsymbol{GST}$
  - Delays bounded by δ
  - Clocks do not drift

Why do we care?

- Real-world applications
  - Blockchain
  - Decentralized Computing
  - IoT

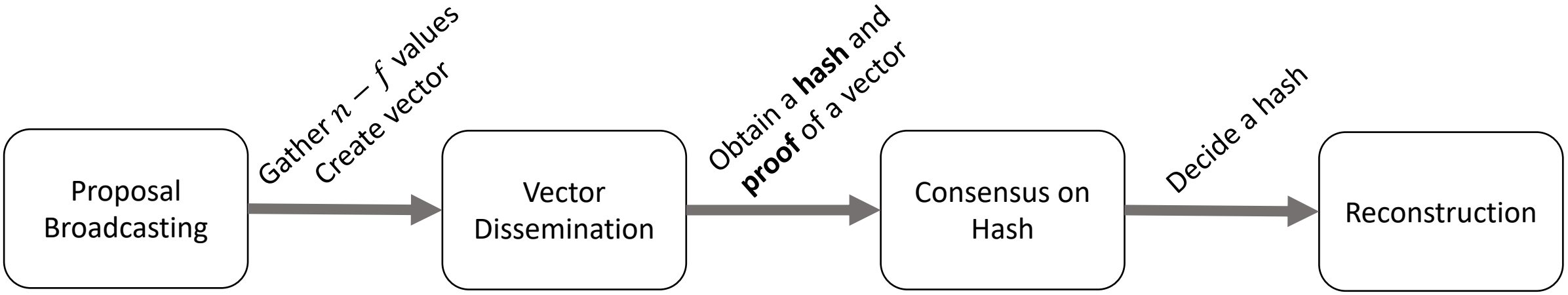




Existing solutions cannot work

- Byzantine Consensus is $O(n^2)$: QUAD
  - For vector consensus: $O(n^3)$ communication complexity, $O(f)$ latency
  - Message size is $O(n)$
- Alternative: Work with hashes instead of vectors
  - $O(n^2 \log(n))$ communication complexity, $O(n^f)$ latency
- Can we do better?
  - We achieved $\boldsymbol{O(n^2\sqrt{n})}$ **communication complexity** and $\boldsymbol{O(n\sqrt{n})}$ **latency**!

Structure of Byzantine Vector Consensus



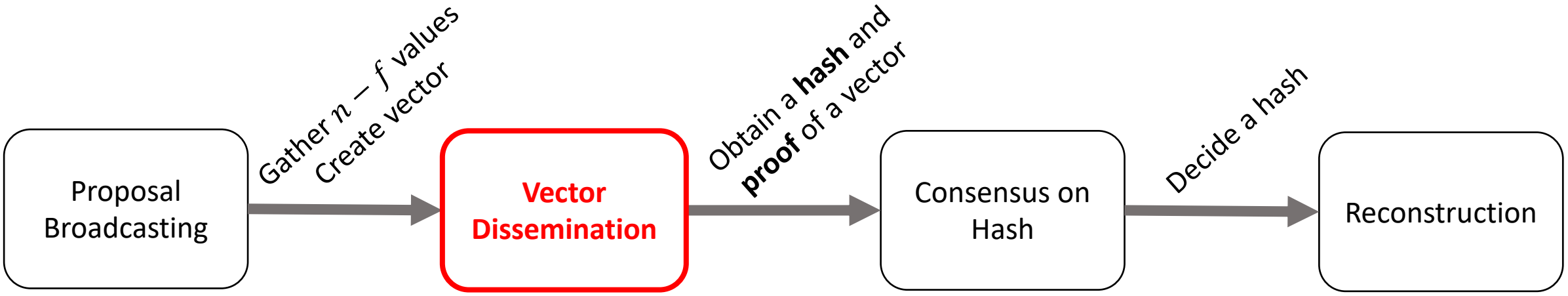
Communication Complexity

$O(n^2)$	???	$O(n^2)$	$O(n^2 \log(n))$

Latency

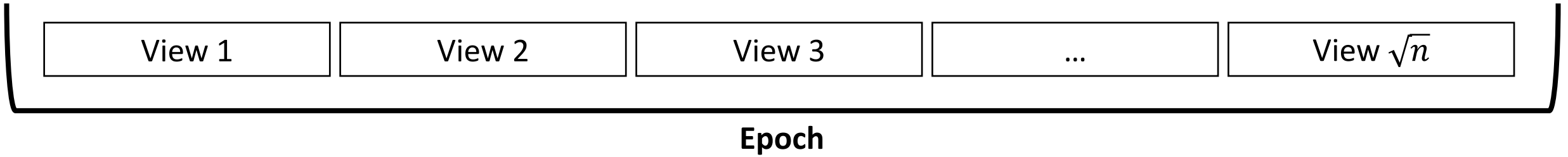
$O(1)$	???	$O(f)$	$O(1)$

Structure of Byzantine Vector Consensus



Communication Complexity	$O(n^2)$	$\boldsymbol{O(n^2\sqrt{n})}$	$O(n^2)$	$O(n^2 \log(n))$
Latency	$O(1)$	$\boldsymbol{O(n\sqrt{n})}$	$O(f)$	$O(1)$

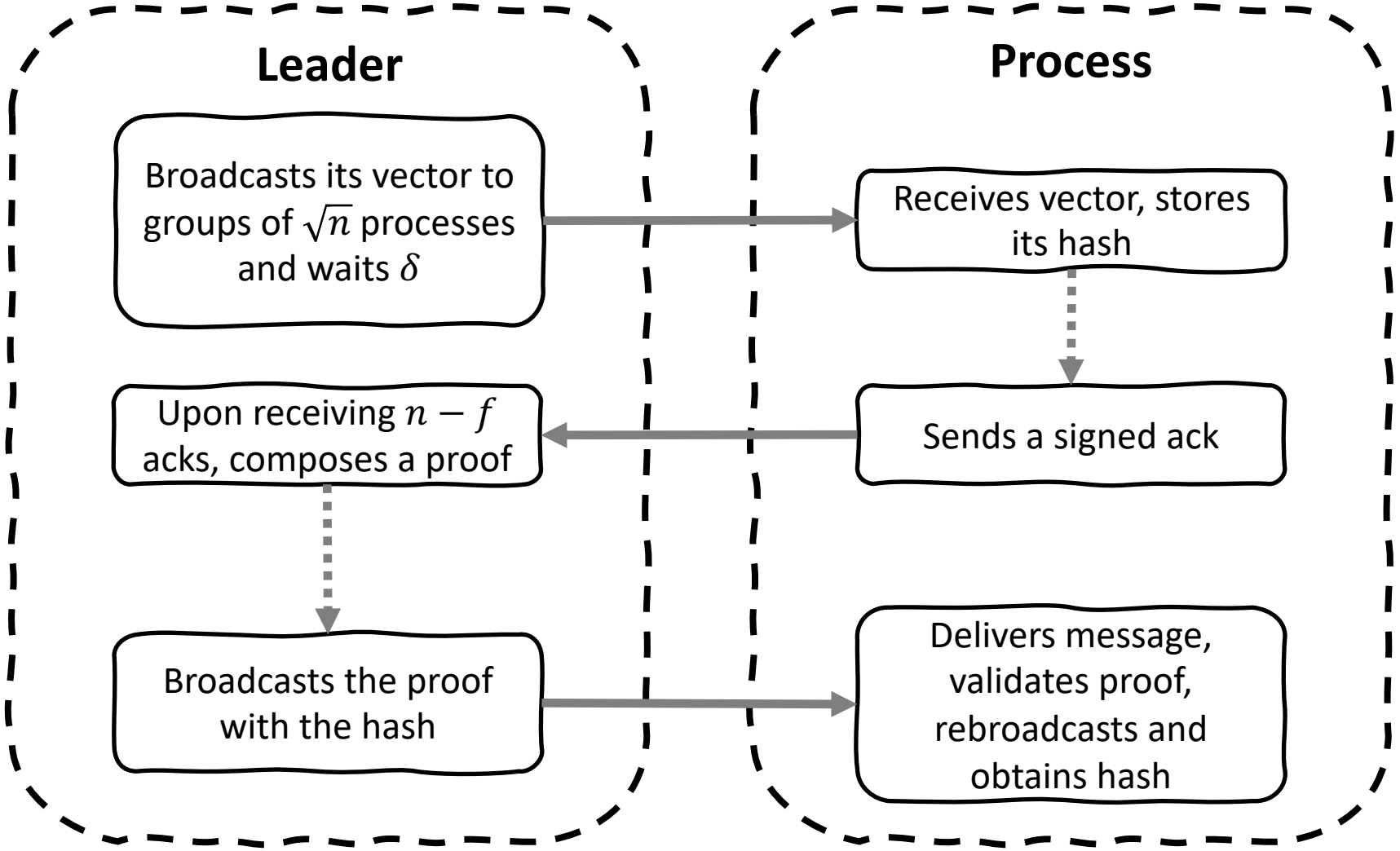
Leader-Based Vector Dissemination



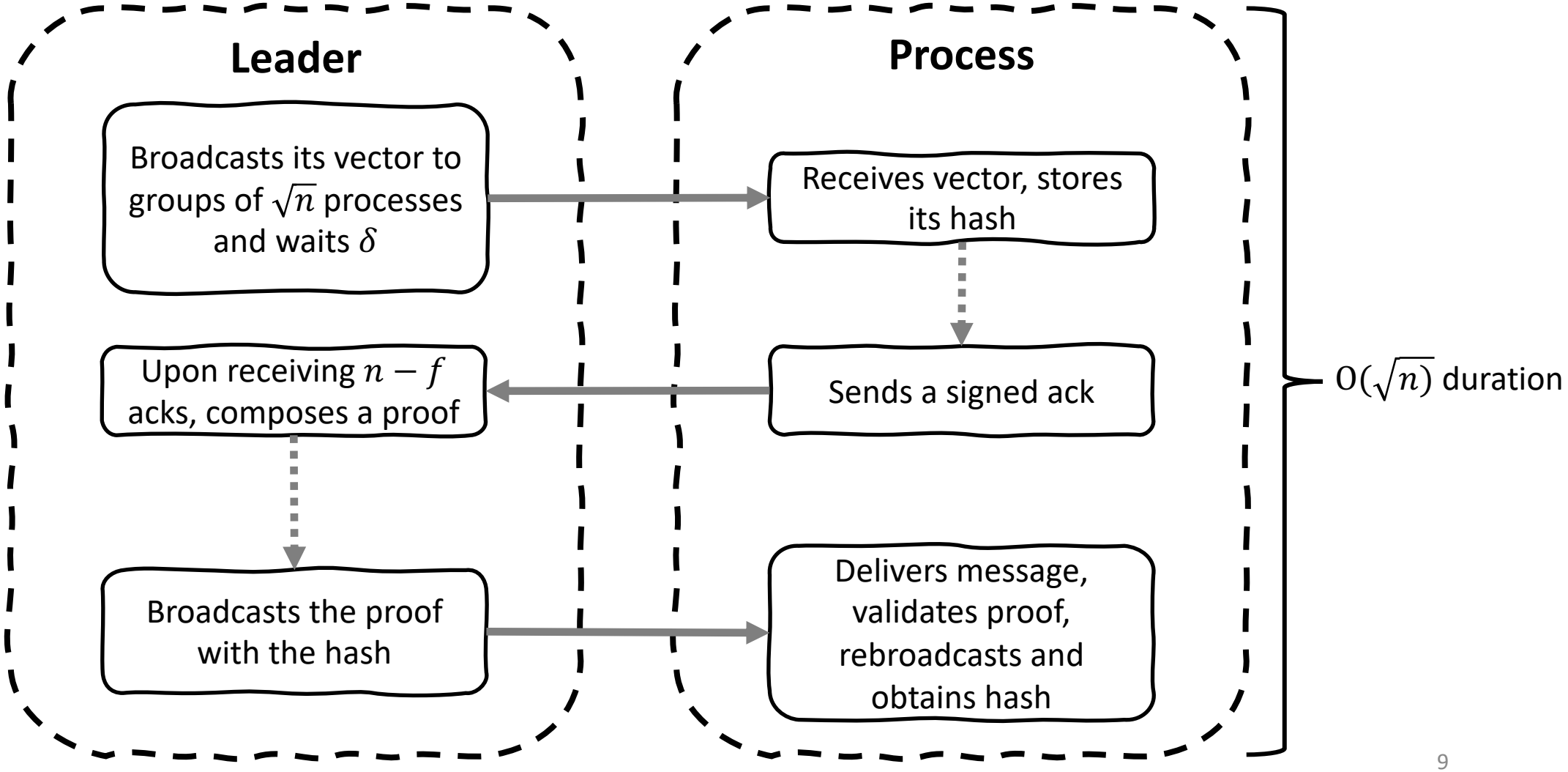
- Each view has a leader process that tries to disseminate their vector (**View Core**)

- Hash obtained when all processes overlap for enough time in a view with correct leader (synchronization)
  - Processes advance over views until they synchronize (**View Synchronizer**)

View Core


Leader
Broadcasts its vector to groups of $\sqrt{n}$ processes and waits δ
Upon receiving $n - f$ acks, composes a proof
Broadcasts the proof with the hash
Process
Receives vector, stores its hash
Sends a signed ack
Delivers message, validates proof, rebroadcasts and obtains hash

View Core

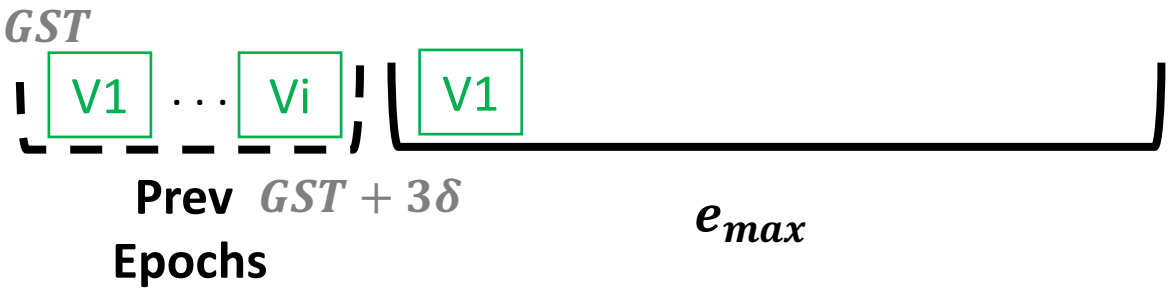

Leader
Process
Broadcasts its vector to groups of $\sqrt{n}$ processes and waits δ
Receives vector, stores its hash
Upon receiving $n - f$ acks, composes a proof
Sends a signed ack
Broadcasts the proof with the hash
Delivers message, validates proof, rebroadcasts and obtains hash
$O(\sqrt{n})$ duration

View Synchronizer

- RareSync: Processes overlap for enough time for every view of epochs entered after GST
  - $f + 1$ views ($O(\sqrt{n})$ epochs) to reach a correct leader (worst case)

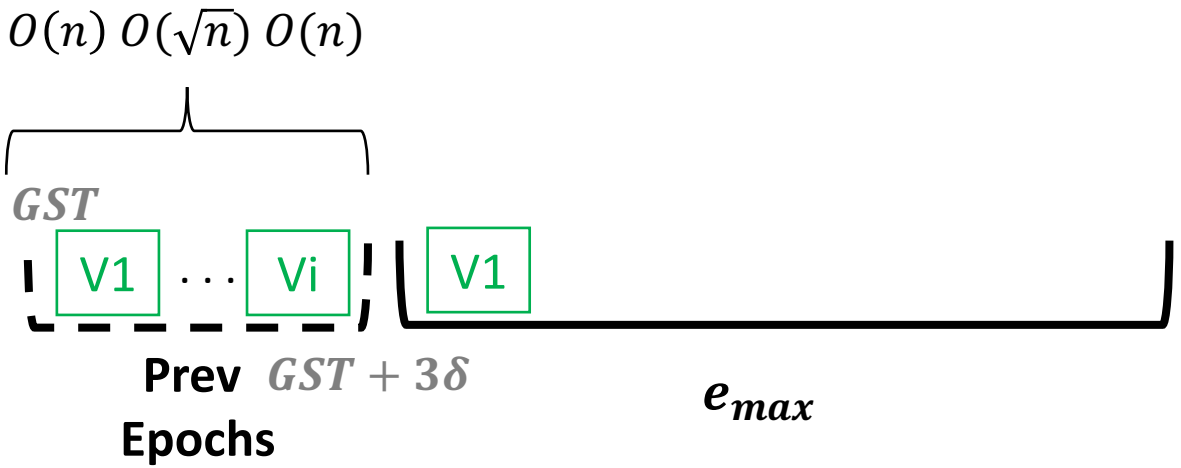
- Processes communicate only at start/end of epochs
  - “all-to-all” communication
  - Local clocks to advance inside epochs
  - Wait δ before entering new epoch

Communication Complexity



- e_{max}: Epoch of most advanced process at GST.
- Worst case:
  - At GST, all correct processes are leaders
  - Most advanced process in e_{max}: All correct processes will enter e_{max} by $GST + 3\delta$
    - All leaders correct but processes misaligned

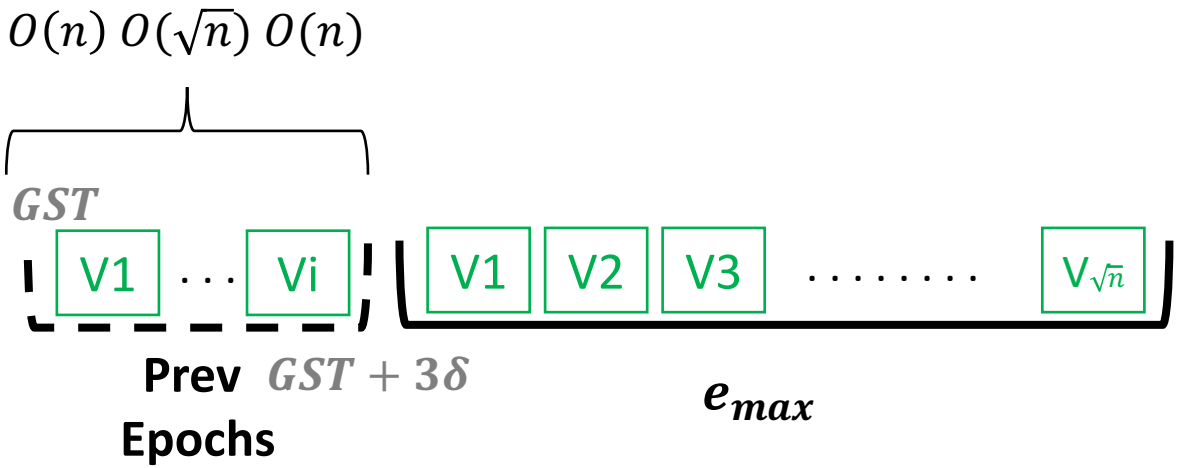
Communication Complexity



- e_{max}: Epoch of most advanced process at GST.

- Worst case:
  - At GST, all correct processes are leaders
  - Most advanced process in e_{max}: All correct processes will enter e_{max} by $GST + 3\delta$
    - All leaders correct but processes misaligned

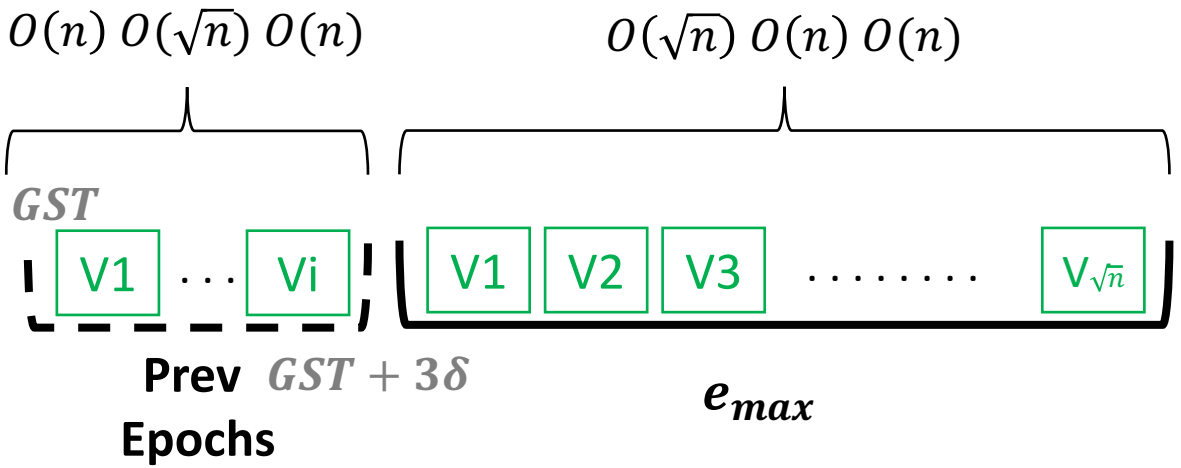
Communication Complexity



- e_{max}: Epoch of most advanced process at GST.

- Worst case:
  - At GST, all correct processes are leaders
  - Most advanced process in e_{max}: All correct processes will enter e_{max} by $GST + 3\delta$
    - All leaders correct but processes misaligned

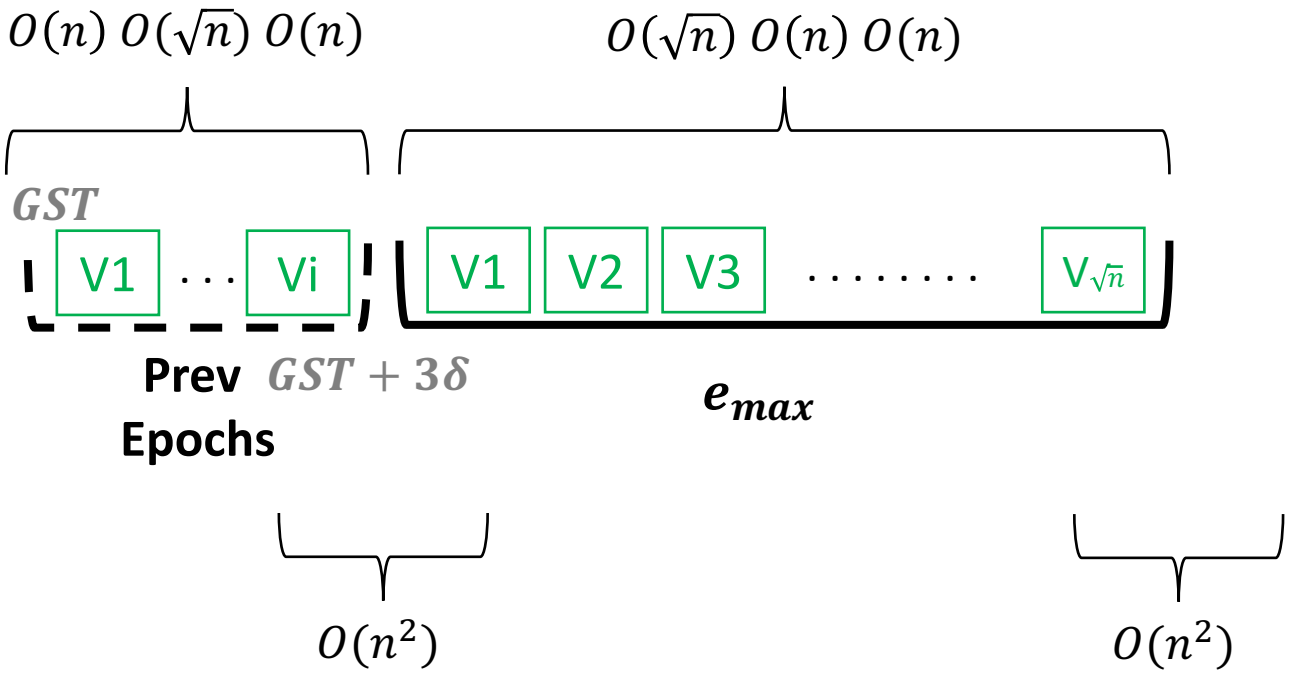
Communication Complexity



- e_{max}: Epoch of most advanced process at GST.

- Worst case:
  - At GST, all correct processes are leaders
  - Most advanced process in e_{max}: All correct processes will enter e_{max} by $GST + 3\delta$
    - All leaders correct but processes misaligned

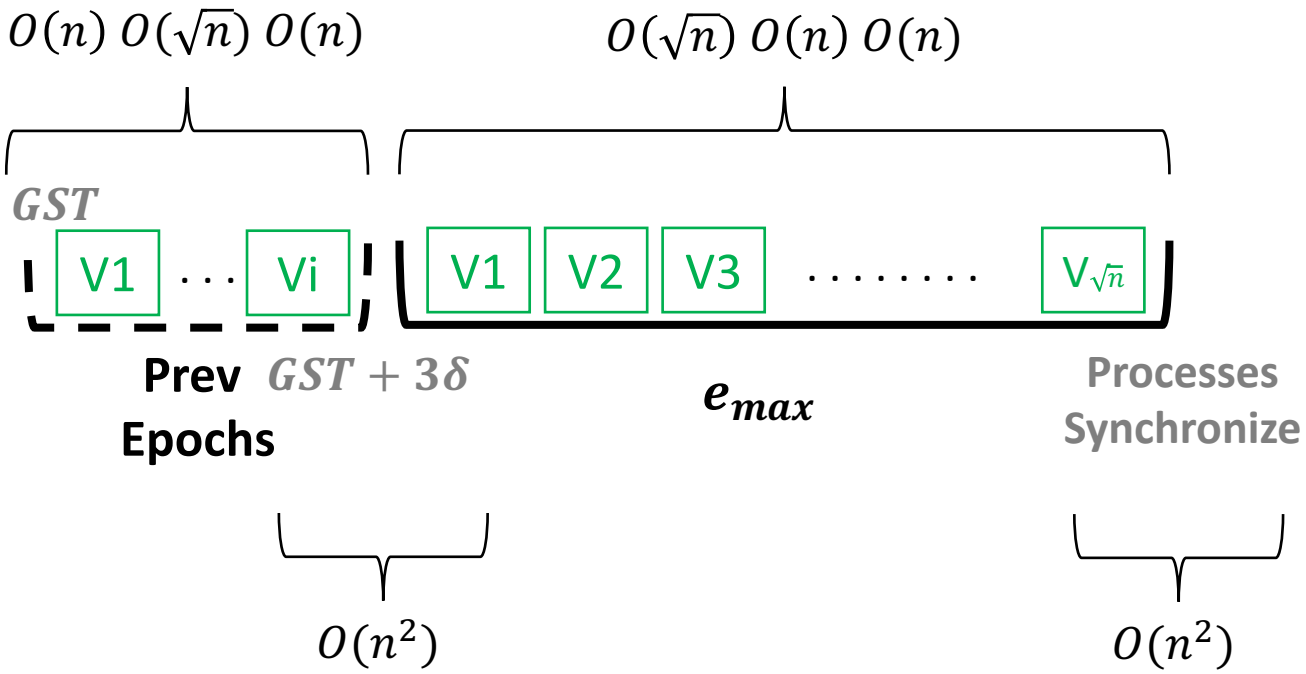
Communication Complexity



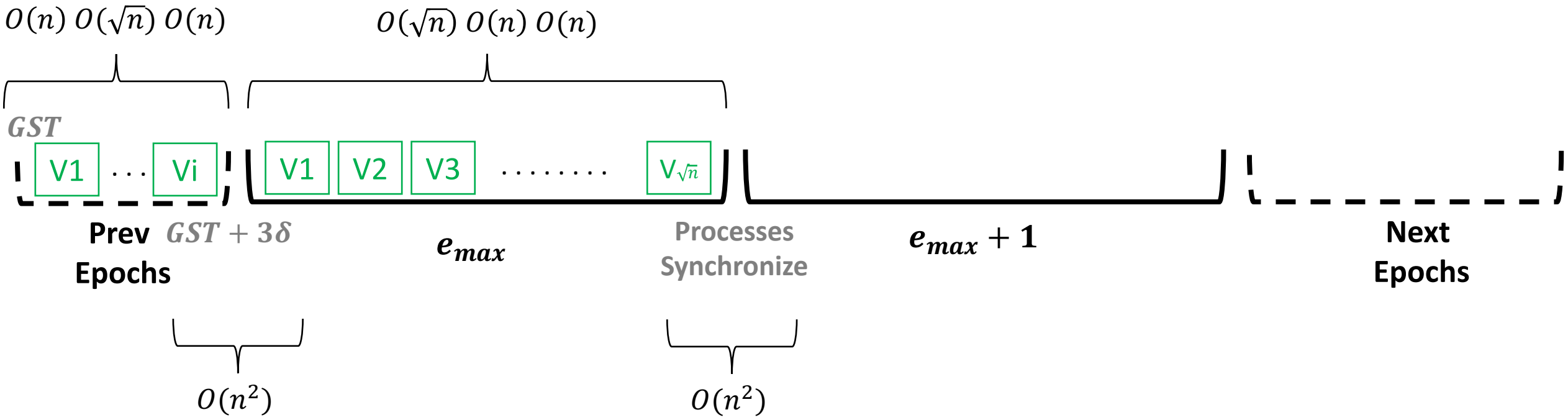
- e_{max}: Epoch of most advanced process at GST.

- Worst case:
  - At GST, all correct processes are leaders
  - Most advanced process in e_{max}: All correct processes will enter e_{max} by $GST + 3\delta$
    - All leaders correct but processes misaligned

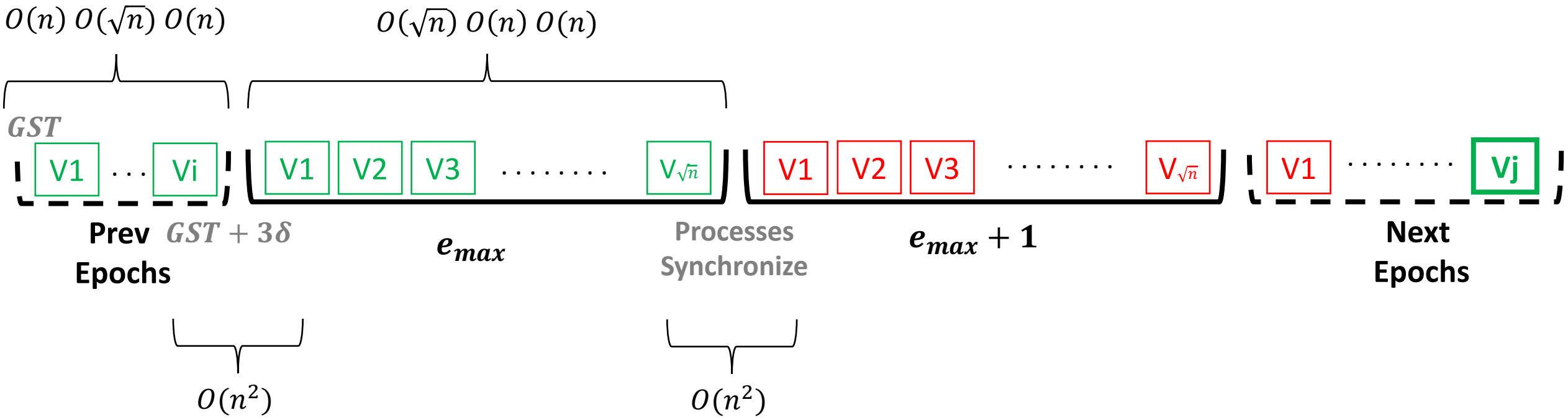
Communication Complexity



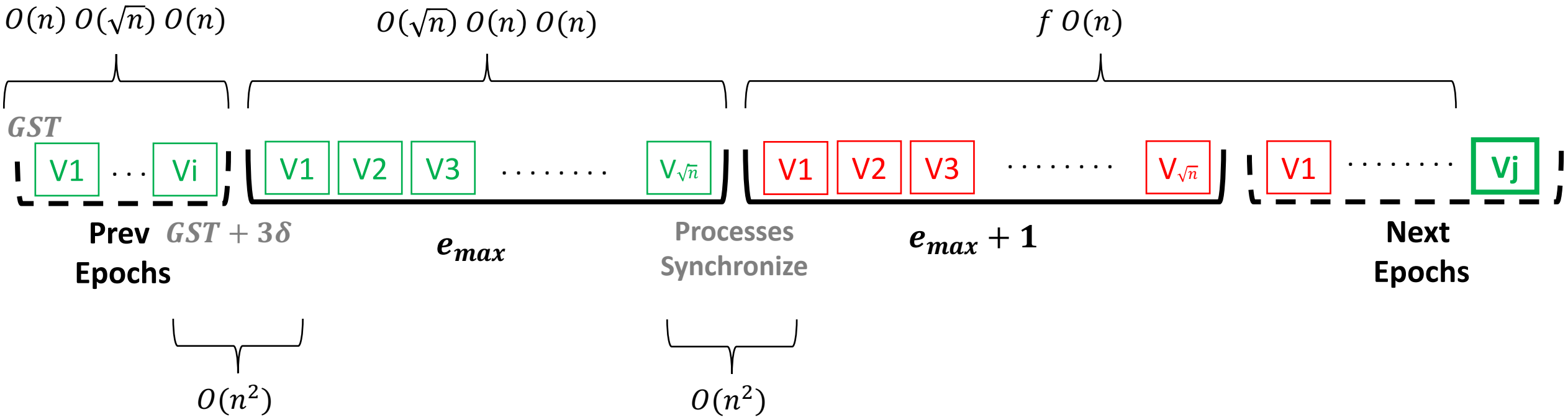
Communication Complexity



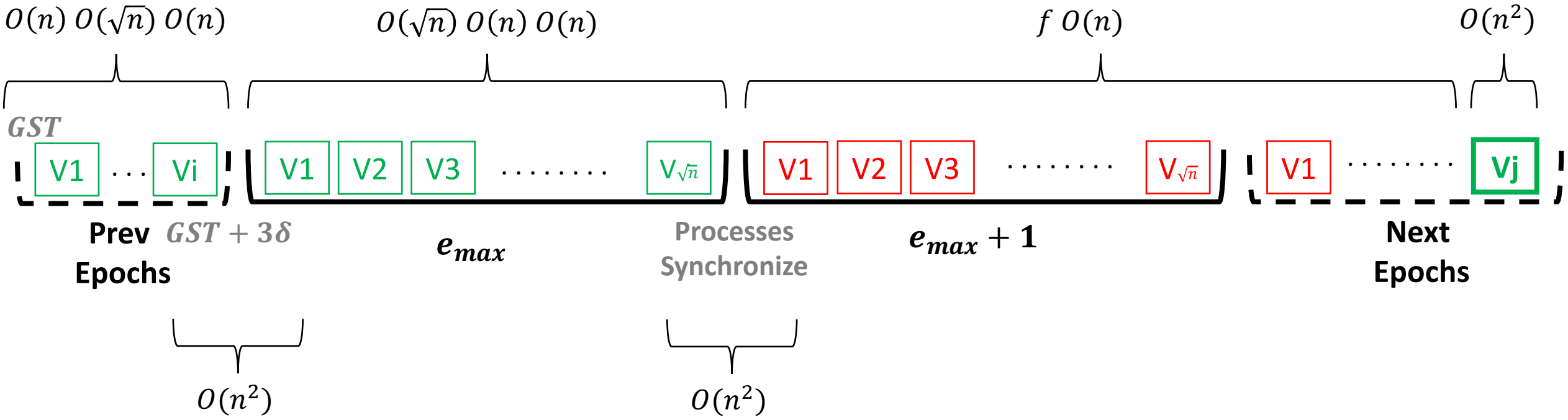
Communication Complexity



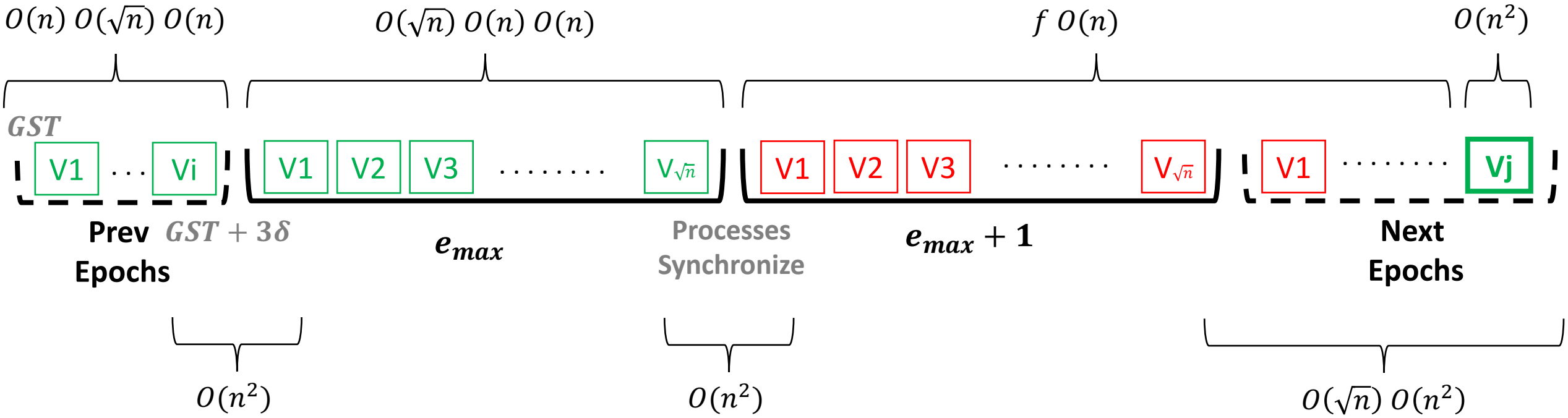
Communication Complexity



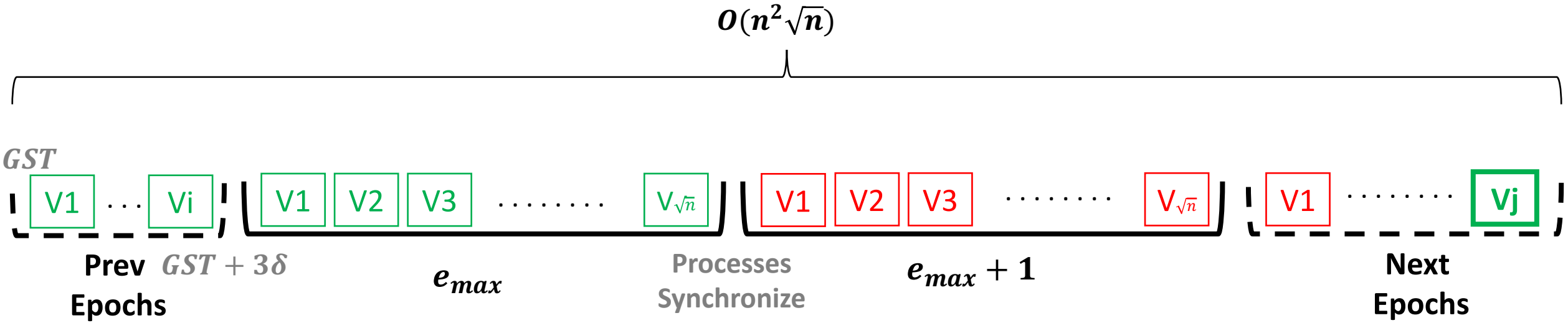
Communication Complexity



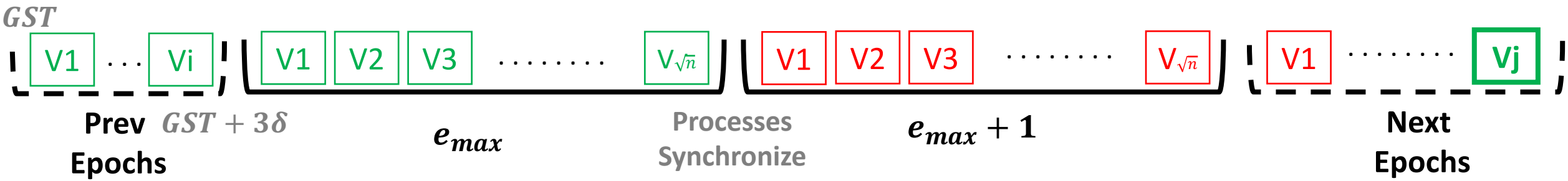
Communication Complexity



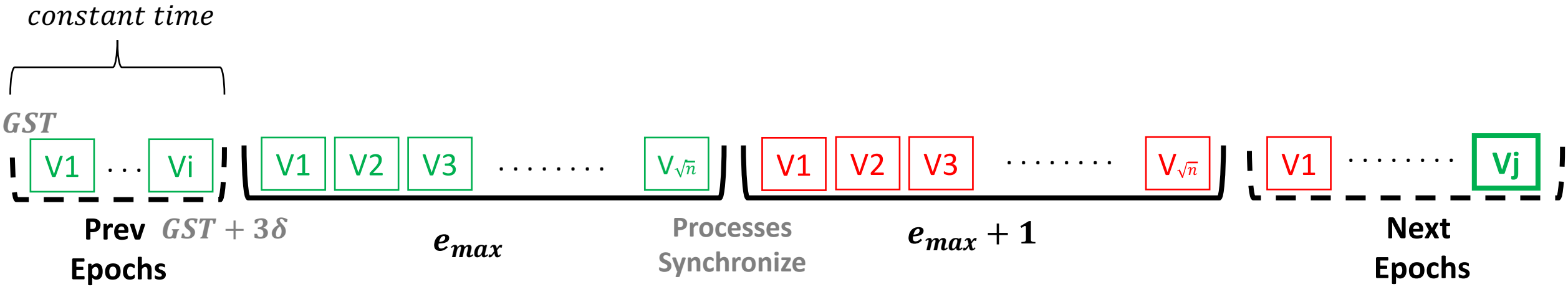
Communication Complexity



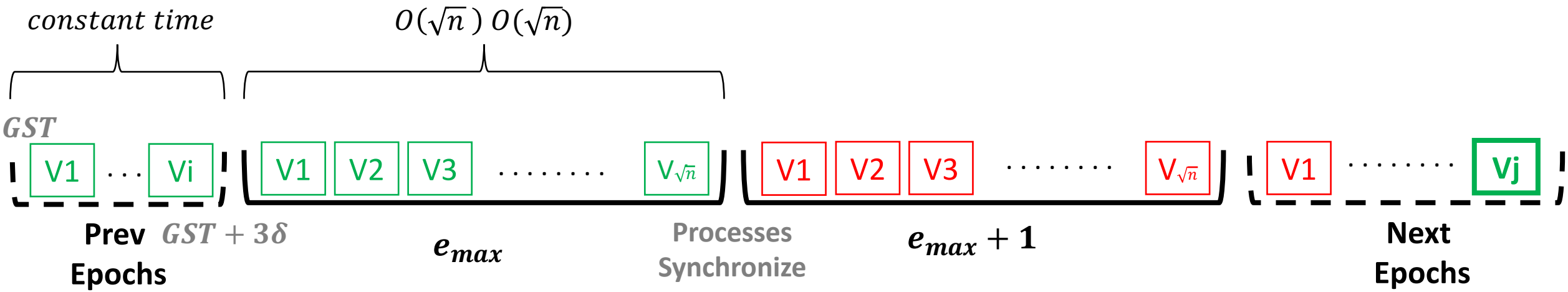
Latency


GST
V1
Vi
V1
V2
V3
V√n
V1
V2
V3
V√n
V1
Vj
Prev Epochs
GST + 3δ
e_max
Processes Synchronize
e_max + 1
Next Epochs

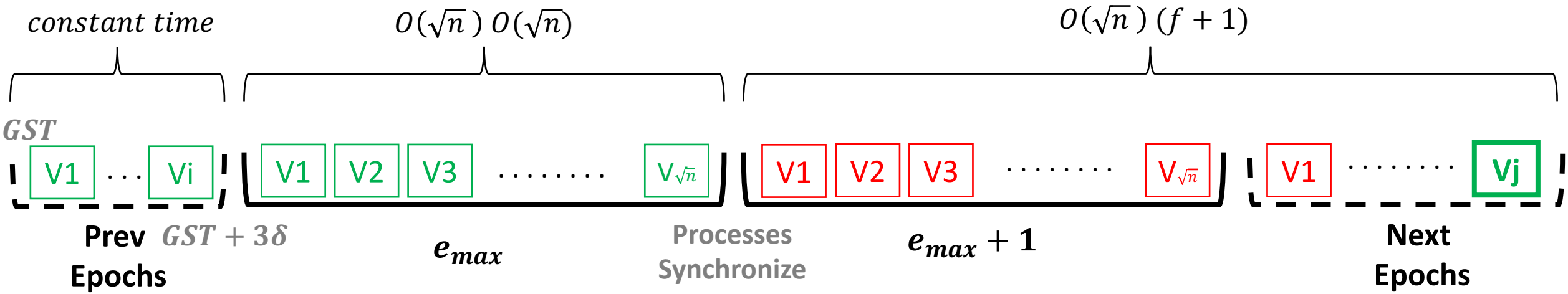
Latency


constant time
GST
V1
Vi
V1
V2
V3
V√n
V1
V2
V3
V√n
V1
Vj
Prev Epochs
GST + 3δ
e_max
Processes Synchronize
e_max + 1
Next Epochs

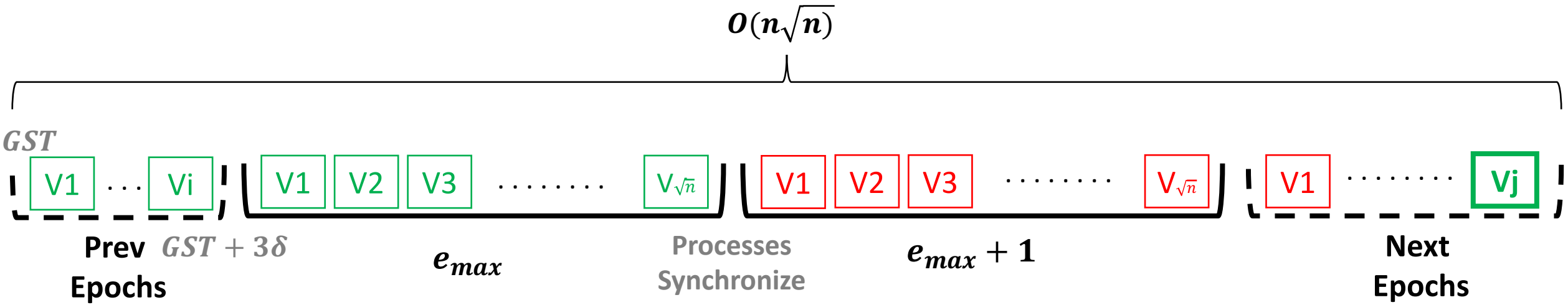
Latency


$constant\ time$
$O(\sqrt{n})$ $O(\sqrt{n})$
GST
V1
Vi
V1
V2
V3
$V_{\sqrt{n}}$
V1
V2
V3
$V_{\sqrt{n}}$
V1
Vj
Prev Epochs
$GST + 3\delta$
e_{max}
Processes Synchronize
$e_{max} + 1$
Next Epochs

Latency

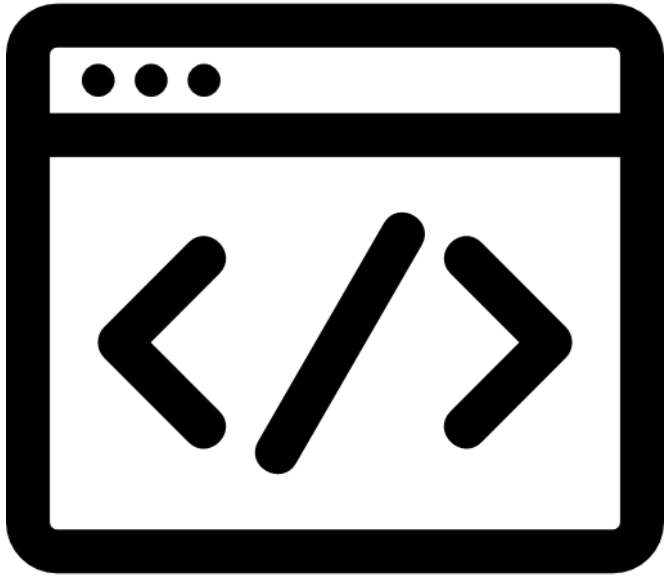

$constant\ time$
$O(\sqrt{n})$ $O(\sqrt{n})$
$O(\sqrt{n})$ $(f+1)$
GST
V1
Vi
V1
V2
V3
$V_{\sqrt{n}}$
V1
V2
V3
$V_{\sqrt{n}}$
V1
Vj
Prev Epochs
$GST + 3\delta$
e_{max}
Processes Synchronize
$e_{max} + 1$
Next Epochs

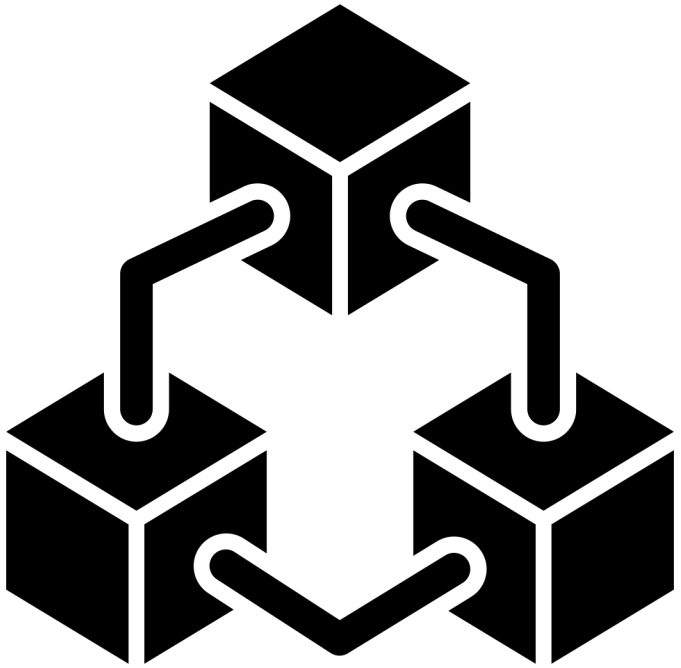
Latency


$O(n\sqrt{n})$
GST
V1
Vi
V1
V2
V3
$V_{\sqrt{n}}$
V1
V2
V3
$V_{\sqrt{n}}$
V1
Vj
Prev Epochs
$GST + 3\delta$
e_{max}
Processes Synchronize
$e_{max} + 1$
Next Epochs

Summary

- Byzantine Vector Consensus is a difficult problem
  - With many real-world applications!
- Previous solutions could not work
  - Either cubic communication complexity or exponential latency
- **Leader-Based Vector Dissemination**
  - Achieves $\boldsymbol{O(n^2\sqrt{n})}$ communication complexity and $\boldsymbol{O(n\sqrt{n})}$ latency!





Thank you!

Questions?

Appendix

Vector Consensus

Problem Definition. In distributed systems, consensus refers to the setting where processes propose an initial value and eventually decide on the same value. Vector consensus allows the processes to agree on a vector of $n - f$ values.

Interface. Vector consensus exposes the interface below:

- Request: propose(Value u). Proposes a scalar value u for consensus.
- Indication: decide(Vector v). Decides a vector v of $n - f$ values.

Properties. Vector consensus exposes the following properties:

1. Agreement: No two correct processes decide differently.
2. Termination: Every correct process eventually decides.
3. Integrity: No process proposes more than once.
4. Vector Validity: Let a correct process decide a vector v of $n - f$ values. Then, every value u of v was proposed by a process P to consensus.

Vector Dissemination

Problem Definition. In vector dissemination, every process proposes its vector of $n - f$ values, and eventually every correct process obtains a hash value and a proof of a vector.

Interface. The module exposes the following interface:

- Request: disseminate(Vector *vec*). Disseminates vector *vec* to the system
- Indication: obtain(Hash_Value H, Storage_Proof sp). Obtains a hash value and a storage proof for a disseminated vector

Properties. Formally, vector dissemination has the following properties:

1. Termination: Every correct process eventually obtains a hash value and a storage proof.
2. Integrity: If a process obtains a hash value H and a storage proof sp, then valid_SP(H,sp)=true, i.e. sp is a valid signature for the disseminated vector.
3. Redundancy: Let a process obtain a storage proof sp which for some hash value H, valid_-SP(H,sp)=true. Then, at least $f+1$ correct processes have cached a vector *vec*, such that $hash(vec) = h$.
4. δ-Closeness: Let t_{first} be the first time a correct process obtains a hash value and a storage proof. Then, every correct process obtains a hash value and a storage proof by time $max(GST, t_{first}) + \delta$.

View Synchronization

Problem Definition. The view synchronization problem is the problem of bringing all processes in the same view with a correct leader for a sufficiently long time. The views are denoted as a set $Views = \{1, 2, 3, ...\}$, and for each view, $u \in Views$ we have a designated leader process denoted as $leader(u)$. View Synchronizers are responsible for advancing the processes through views after keeping them in a view for sufficiently long, till they reach a view with a correct leader.

Interface. Thus, they expose the following interface:

- Indication: advance(View u). The process advances to view u.

We say that a process P enters a view u at time t only if the advance indication occurred at time t. If another advance indication occurs for another view u' at time t', then we say that P remained in view u during $[t, t')$. If no newer advance occurs, then P will remain in view u.

Synchronization time. We now define the notion of Synchronization time, which is the time when all correct processes overlap in the same view for at least Δ time.

Definition 1 (Synchronization time)**.** *Time t_s is a synchronization time if (1) all correct processes are in the same view from time t_s to (at least) time $t_s + \Delta$ and (2) the leader of the view is correct.*

View synchronization ensures the eventual synchronization property which states that there exists a synchronization time at or after GST.

Crypto

We assume a (k,n)-threshold signature scheme [7], where $k = 2f + 1 = n - f$. In this scheme, each process holds a distinct private key and there is a single public key. Each process P_i can use its private key to produce a partial signature of a message m by invoking $ShareSign(m)$. A partial signature $tsignature$ of a message m produced by a process P_i can be verified by $ShareVerify(m, tsignature)$. Finally, set $S = \{tsignature\}$ of partial signatures, where $|S| = k$ and, for each $tsignature_i \in S$, $tsignature = ShareSign(m)$, can be combined into a single (threshold) signature by invoking $Combine(S)$. The produced threshold signature has the same size as the partial ones. A Byzantine process cannot produce a valid threshold signature with less than $k = n - f = 2f + 1$ partial signatures. We suppose that we have a provided $valid\_SP$ function, that given a threshold signature, responds $\{true, false\}$ based on if the signature is valid or not.

Byzantine Vector Consensus

Algorithm 1 Vector consensus pseudocode for process P_i

1: **Uses**:
2: Best-effort broadcast, instance *beb*
3: Vector dissemination, instance *dissemination*, of section 5, with $|VE|$ views per epoch.
4: QUAD, instance *quad* [1]
5: ADD, instance *add* [3]
6: **upon** init:
7: Integer $received\_proposals \leftarrow 0$
8: Map(Process $\rightarrow$ Value) $proposals_i \leftarrow empty$
9: Map(Process $\rightarrow$ Message) $messages_i \leftarrow empty$
10: **upon** propose(Value u):
11: Signature $sig_i \leftarrow$ signature of message $\{PROPOSAL, u\}$
12: **invoke** $beb.broadcast(sig_i)$
13: **upon** reception of sig_j of message $\{PROPOSAL, u_j\}$ from P_j and $received\_proposals_i < n - f$:
14: $received\_proposals_i \leftarrow received\_proposals_i + 1$
15: $proposals_i[P_j] \leftarrow u_j$
16: $messages_i[P_j] \leftarrow sig_j$
17: **if** $received\_proposals = n - f$:
18: Vector $vec \leftarrow$ Create an input configuration vector from the $proposals_i$
19: $disseminator.disseminate(vec)$
20: **upon** $disseminator.obtain$(Hash_Value H, Storage_Proof sp):
21: **if** not yet proposed to *quad*:
22: **invoke** $quad.propose((H, sp))$
23: **upon** $quad.decide$((Hash_Value H', Storage_Proof sp')):
24: Vector $vec' \leftarrow$ a cached vector with H' hash value
25: **invoke** add.input(vec')
26: **upon** $add.output$(Vector vec''):
27: **trigger** $decide(vec'')$

Leader-Based Vector Dissemination

Algorithm 2 Leader-based vector dissemination pseudocode for process P_i

```
1:  Modules
2:      View_Core core ← View Core with batch size |VE|
3:      View_Synchronizer synchronizer ← Generalized RareSync with |VE| epochs
4:  upon disseminate(Vector v):
5:      core.init(v)
6:      synchronizer.init
7:  upon synchronizer.advance(View u):
8:      core.start_executing(u)
9:  upon core.obtain(Hash_Value H, Storage_Proof sp):
10:     trigger obtain(H,sp)
```

View Core

Algorithm 3 View Core pseudocode for view u and process P_i

```
1:  Uses
2:      Best-Effort Broadcast, instance beb
3:  upon init(Vector v):
4:      Map (Hash_Value → Vector) vectors_i ← empty
5:      Set (Process) disseminated_i ← empty
6:      Hash_Value H_i ← None
7:      if process P_i is leader(u)
8:          H_i ← hash(v)
9:          while vector v is not broadcasted to all processes
10:             invoke beb.broadcast({PROPOSE, v}) to |VE| processes
11:             wait δ time
12: upon beb.deliver(Message {PROPOSE, v'}, Process P_j):
13:     if process P_j ∉ disseminated_i
14:         disseminated_i ← disseminated_i ∪ { P_j }
15:         vectors_i[hash(v')] ← v'
16:         Signature sig ← Signature of the message {STORED, hash(v')}
17:         send sig to P_j
18: upon reception of sig of message {STORED, H'} such that H' = H_i from n − f distinct processes:
19:         Storage_Proof sp ← Combine({sig | sig is a signature of a received STORED message})
20:         invoke beb.broadcast({DECIDE, H_i, sp})
21: upon reception of message {DECIDE, H', sp'}:
22:     if sp' is a valid n − f threshold signature
23:         invoke beb.broadcast({DECIDE, H', sp'})
24:         trigger obtain(H', sp')
25:         stop participating in vector dissemination
```

System Model

Processes. Our system consists of a static set of n processes, which we denote as $\{P_1, P_2, ..., P_n\}$. Our Byzantine distributed context assumes that there might be up to f faulty processes, where $n = 3f + 1$. Faulty processes can behave in an arbitrary way. For any process that is not faulty, we say that it is correct. Processes may communicate through authenticated point-to-point links, and the communication between two processes is reliable: If a correct process sends a message to another correct process, the message is eventually received. We assume that processes have local hardware clocks. Furthermore, we assume that local steps of processes take zero time, as the time needed for local computation is negligible compared to message delays.

Partial Synchrony. We consider a partially synchronous model as introduced in [5]. For every execution of an algorithm in such a model, there exists a Global Stabilization Time (GST), which is unknown to the processes. Before GST, message delays are unbounded and local clocks may drift arbitrarily, but after GST message delays are bounded by a known delay δ, and the clocks do not drift.

RareSync

RareSync. RareSync [1] is a View Synchronizer with $O(f)$ latency and $O(n^2)$ worst-case communication complexity. It is designed to solve view synchronization, which is the problem of bringing the processes in the same view with a correct leader for a sufficient amount of time. RareSync achieves such results by grouping views into epochs, with $f+1$ views per epoch, and manages to accomplish view synchronization in a constant number of epochs after GST, which is the time when the message delays become bounded by a known time δ.

The main innovation of RareSync is that processes rely on local clocks to proceed over views, and they only communicate at the end of an epoch. When a process finishes the last view of an epoch, it informs the rest of the system by broadcasting an epoch completion ($EPOCH\text{-}COMPLETED$) message. Upon receiving $n-f$ such messages from other processes for the same epoch, and upon waiting δ time, each process enters the new epoch and informs all the other processes by broadcasting an enter epoch ($ENTER\text{-}EPOCH$) message. When processes that are currently in an epoch e deliver such a message for an epoch $e' > e$, they wait for δ time and then they also enter epoch e'. The reason for this δ time wait is to allow processes to receive possible broadcasts for more advanced epochs.

RareSync analysis proves that this method allows the processes will synchronize for at least Δ time in every view of the first epoch entered after GST. This implies that by using RareSync, synchronization can be achieved in a constant number of epochs (and views) after GST.

QUAD

QUAD. QUAD [1] is a partially synchronous Byzantine consensus protocol that exposes the following interface:

1. Request: propose(Value u). Proposes a value u for consensus.
2. Indication: decide(Value v). Decides a value v from consensus.

The protocol was initially proposed for Byzantine consensus, where the messages contain single proposals and have constant size, $O(1)$. In such settings, QUAD is able to achieve consensus in $O(n^2)$, with linear latency of $O(f)$. In addition, QUAD can be adapted for Byzantine vector consensus, but unfortunately the linear size of messages that contain $n - f$ values, $O(n)$ result in protocols that achieve consensus in $O(n^3)$.

Existing Solution

Vector dissemination consensus. The cubic communication complexity of Authenticated vector consensus can be reduced by exploiting a Slow Broadcast-Based vector dissemination. The algorithm achieves $O(n^2 \log n)$ communication complexity, but it is impractical due to the latency of $O(n^f)$. Instead of proposing vectors to QUAD, the processes broadcast their values to the system. When a process gathers $n - f$ proposals, it creates an input configuration vector and initiates the Vector Dissemination protocol. Each correct process eventually obtains a hash value H and a storage proof sp for a vector. Then, the processes propose the pair (H,sp) to QUAD, which is a message of $O(1)$ size. Thus, QUAD responds with a selected pair. Having obtained the selected hash value of the decided vector, which is the same for each correct process due to the agreement property of consensus, the last part of the algorithm initiates the ADD protocol [3], an $O(n^2 \log n)$ protocol for asynchronous data dissemination that ensures that every process will obtain the vector that corresponds to the decided hash value.

The communication complexity of the algorithm is the sum of the communication complexities of the initial broadcast, the Vector Dissemination, QUAD, and ADD, which result in $n \cdot O(n) + O(n^2) + O(n^2) + O(n^2 \log n) = O(n^2 \log n)$. Due to the usage of Vector Dissemination, the latency of the algorithm is $O(n^f)$. A concrete analysis of the algorithm, along with proofs and pseudocode is presented in [2].

Resources

https://github.com/MChatzakis/Byzantine-Vector-Consensus