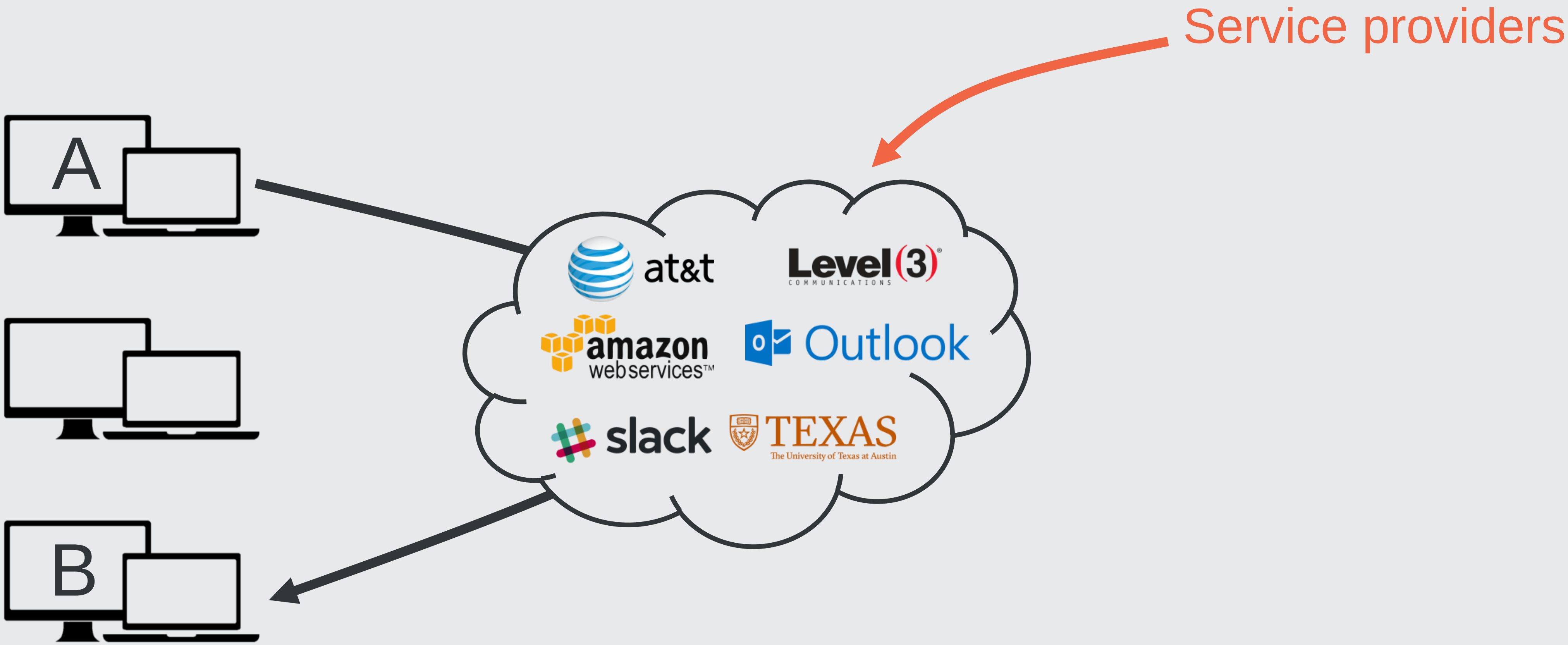
Unobservable communication over fully untrusted infrastructure

Sebastian Angel
UT Austin and NYU

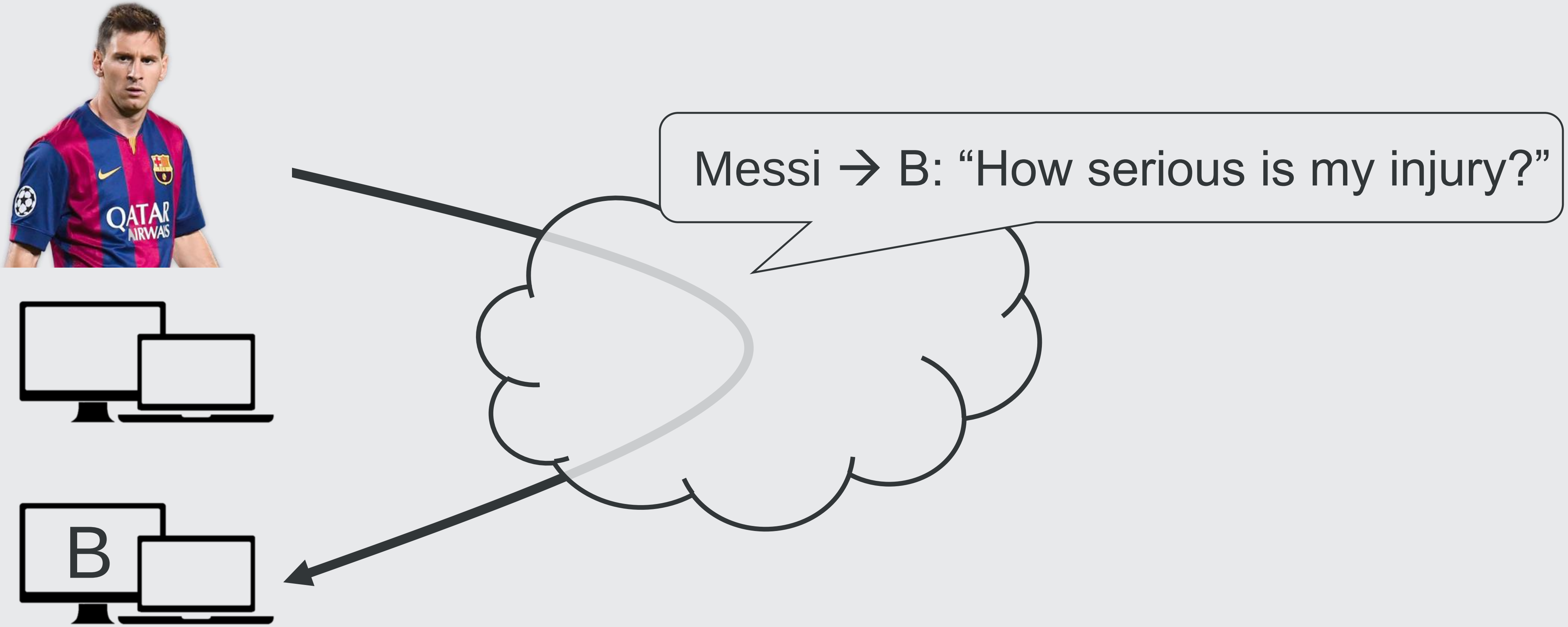
Srinath Setty
Microsoft Research

Communication is possible because of many service providers



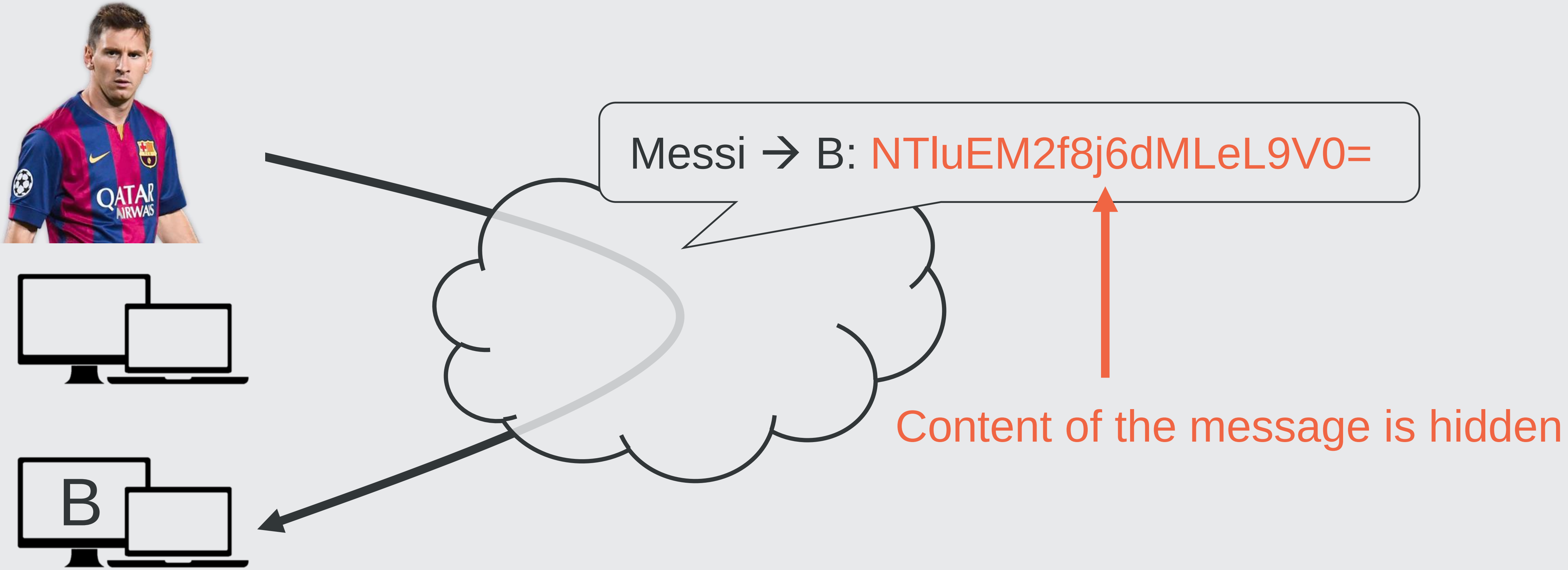
Service providers

These providers can observe all communication



Messi → B: “How serious is my injury?”

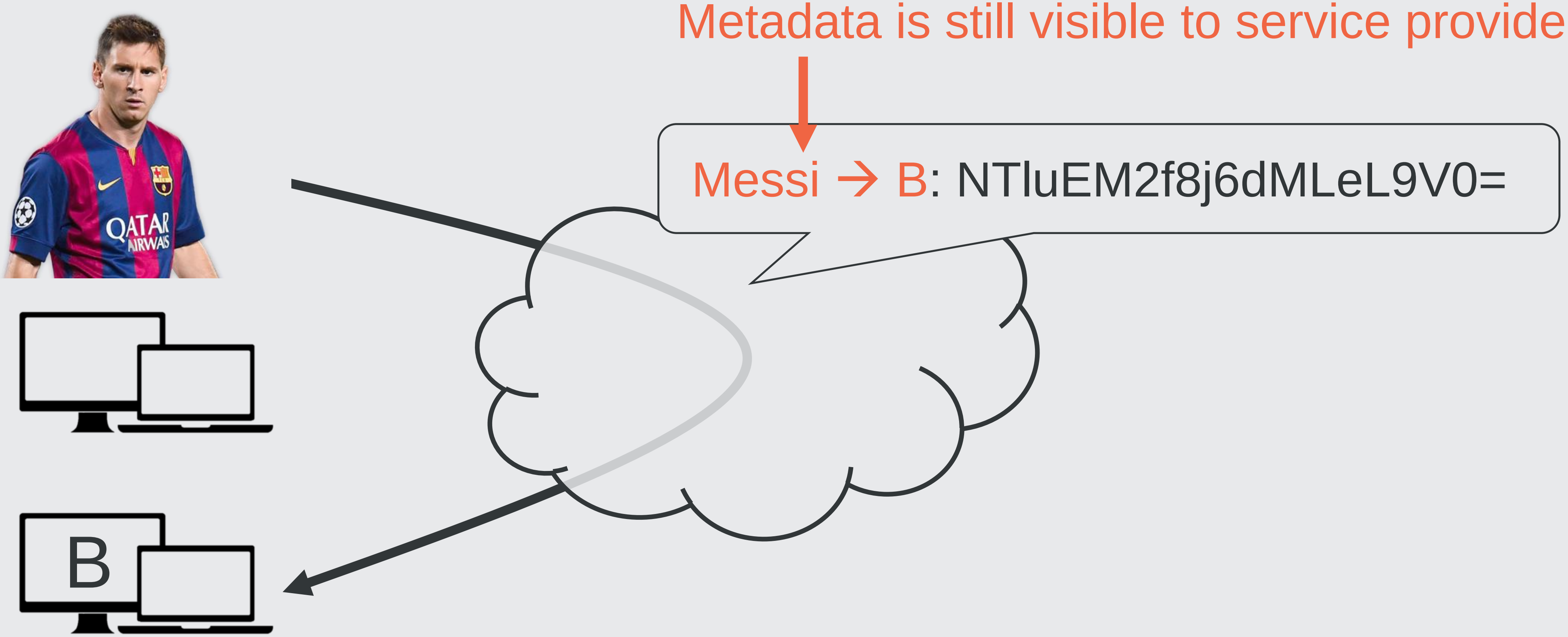
Encryption can hide the message



Messi → B: NTluEM2f8j6dMLeL9V0=

Content of the message is hidden

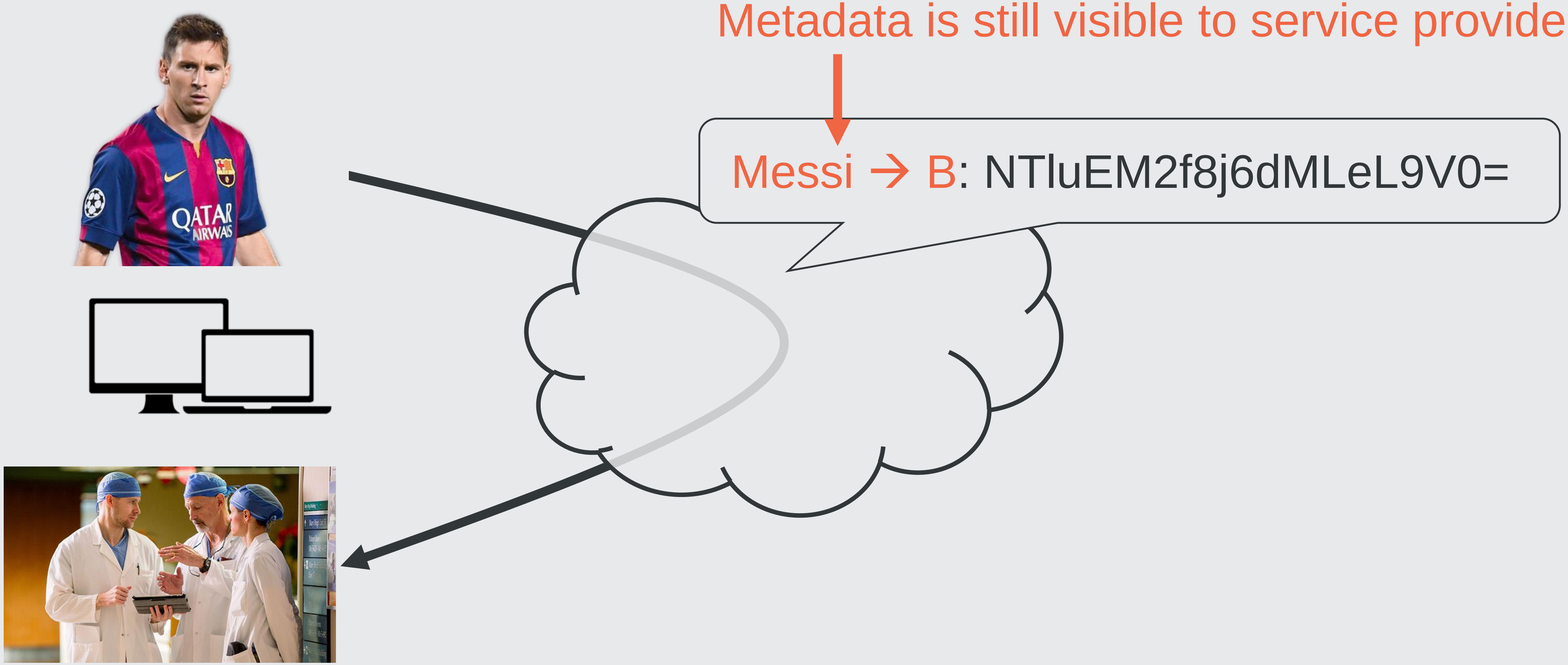
But metadata remains



Metadata is still visible to service providers

Messi → B: NTluEM2f8j6dMLeL9V0=

But metadata remains



Metadata is still visible to service providers

Messi → B: NTluEM2f8j6dMLeL9V0=

Metadata can be as sensitive as data

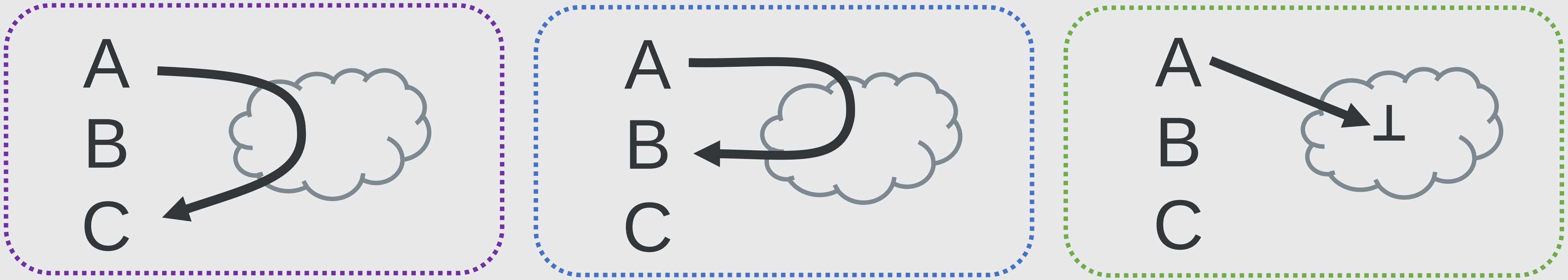
“telephone metadata… can be used to determine highly sensitive traits.”

[Mayer, Mutchler, and Mitchell, PNAS 2016]

General Hayden: “We kill people based on metadata.”
(former NSA and CIA director)

[David Cole, NYR Daily 2014]

Objective: adversary cannot determine who is talking to whom, or if anybody is talking at all

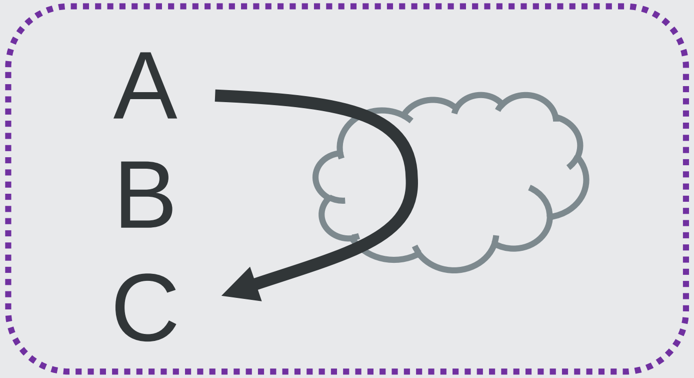


A talks to C

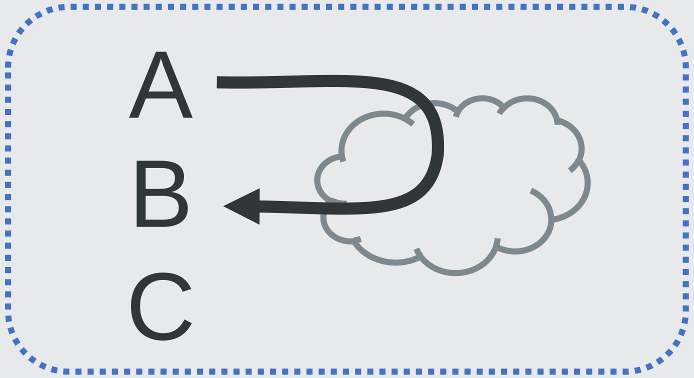
A talks to B

A talks to nobody (“⊥”)

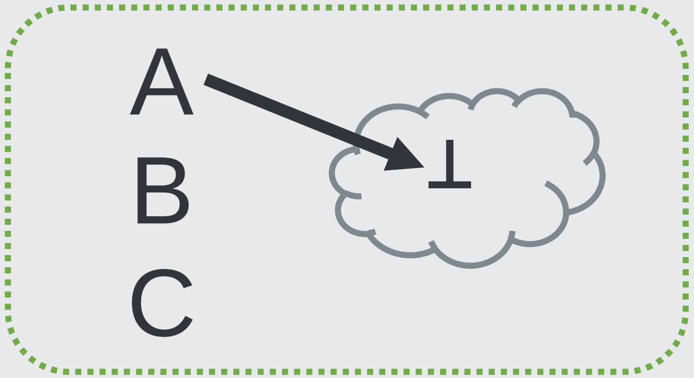
Objective: adversary cannot determine who is talking to whom, or if anybody is talking at all



A talks to C



A talks to B

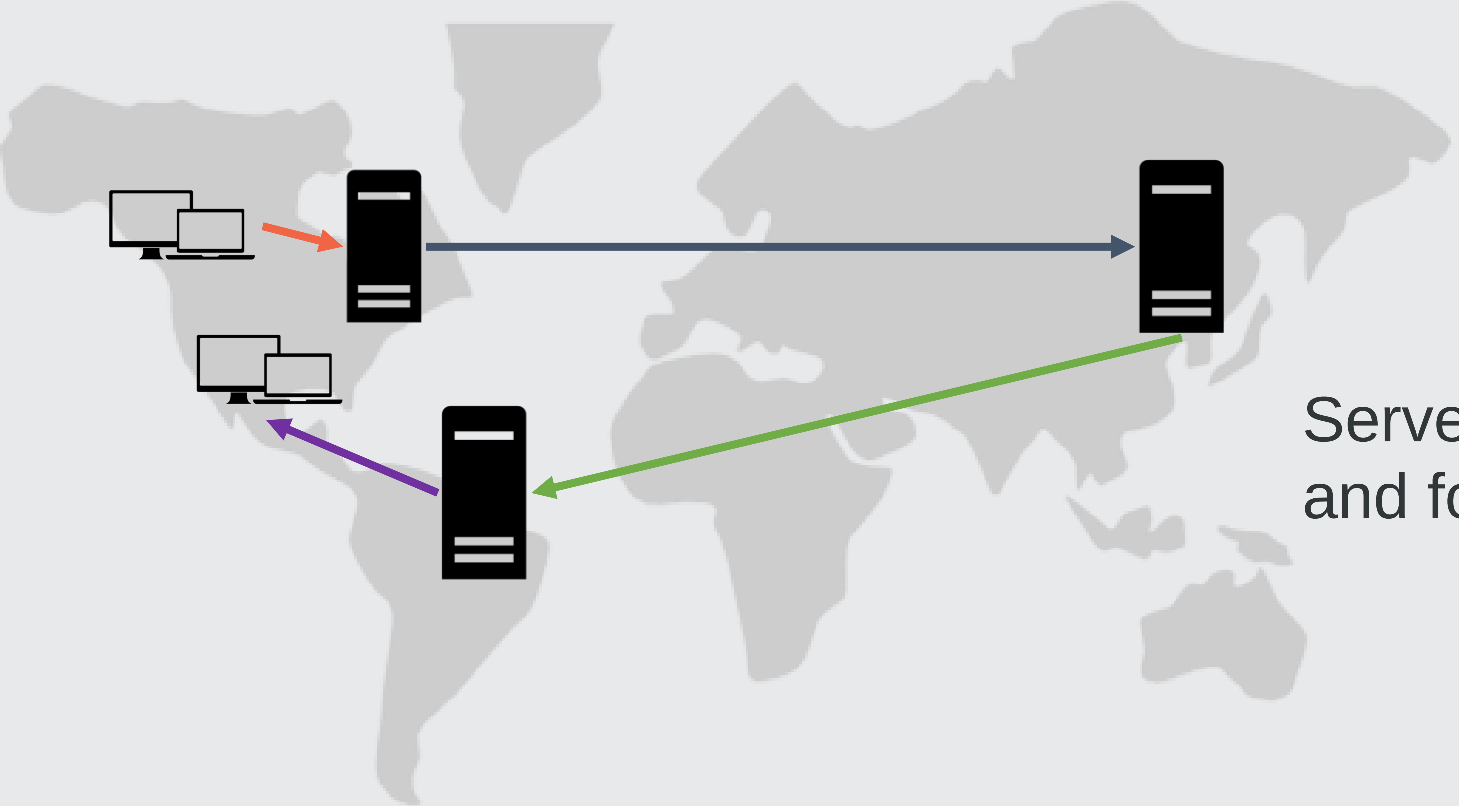


A talks to nobody (“⊥”)

Variants of this objective date back to the 80s [Chaum, CACM ‘81]

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec '04])

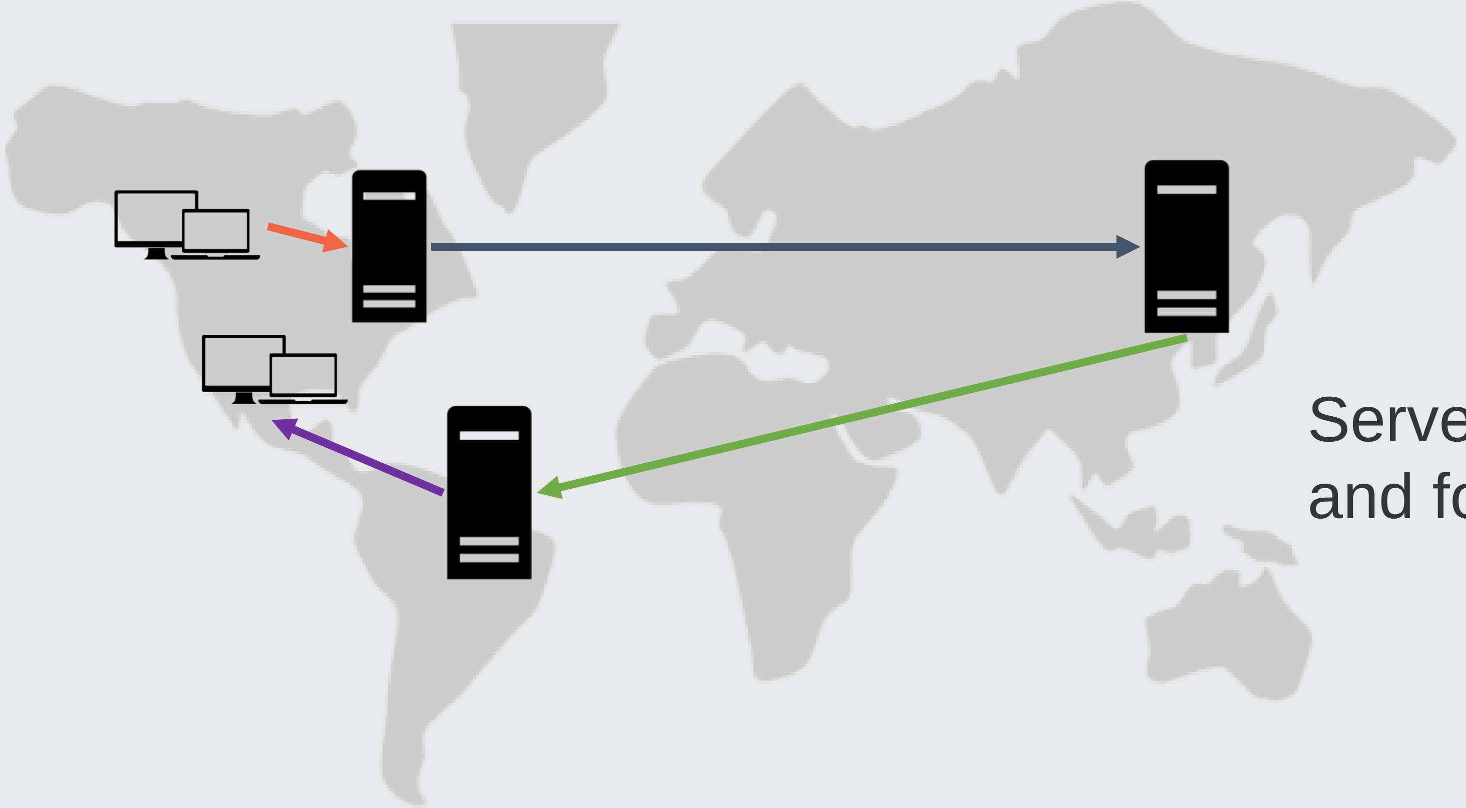


Servers remove one layer of encryption and forward messages to the next hop

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec ‘04])

Strong assumptions on which parts of the infrastructure can be compromised



Servers remove one layer of encryption and forward messages to the next hop

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec ‘04])

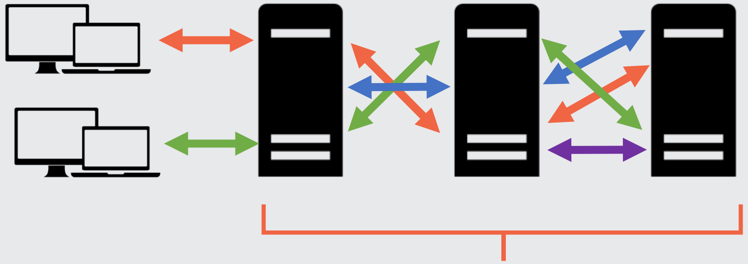
  Supports millions of users but tolerates few compromises

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec ‘04])

  Supports millions of users but tolerates few compromises

- Mix networks (e.g., Vuvuzela [SOSP ’15])



Requires at least one correct server

Servers shuffle traffic, add noise (cover traffic), remove layers of encryption, etc.

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec ‘04])

  Supports millions of users but tolerates few compromises

- Mix networks (e.g., Vuvuzela [SOSP ’15])

  Supports 2 million users but requires one correct server

Many systems already meet this objective!

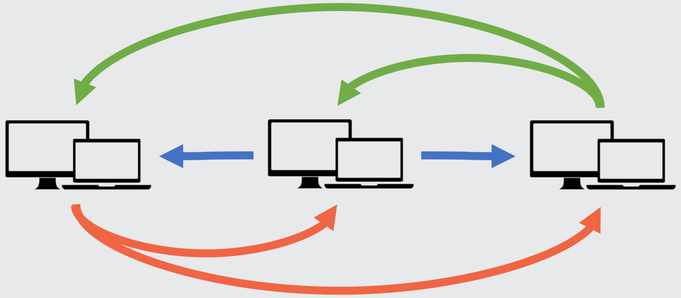
- Onion routing (e.g., Tor [USENIX Sec ‘04])

  Supports millions of users but tolerates few compromises

- Mix networks (e.g., Vuvuzela [SOSP ’15])

  Supports 2 million users but requires one correct server

- DC Networks (e.g., Dissent [CCS ’10])



Peer-to-peer network

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec ‘04])

  Supports millions of users but tolerates few compromises

- Mix networks (e.g., Vuvuzela [SOSP ’15])

  Supports 2 million users but requires one correct server

- DC Networks (e.g., Dissent [CCS ’10])

  Supports dozens of users but tolerates full infrastructure compromise

Many systems already meet this objective!

- Onion routing (e.g., Tor [USENIX Sec ‘04])

  Supports millions of users but tolerates few compromises

- Mix networks (e.g., Vuvuzela [SOSP ’15])

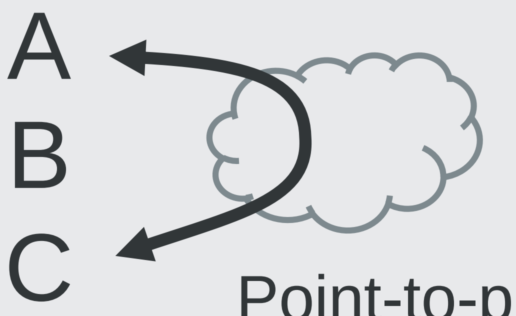
  Supports 2 million users but requires one correct server

- DC Networks (e.g., Dissent [CCS ’10])

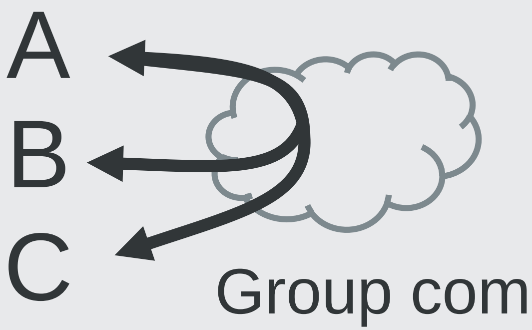
  Supports dozens of users but tolerates full infrastructure compromise

We propose Pung

- Provably hides metadata even if all infrastructure is compromised
- Supports point-to-point and group communication



Point-to-point



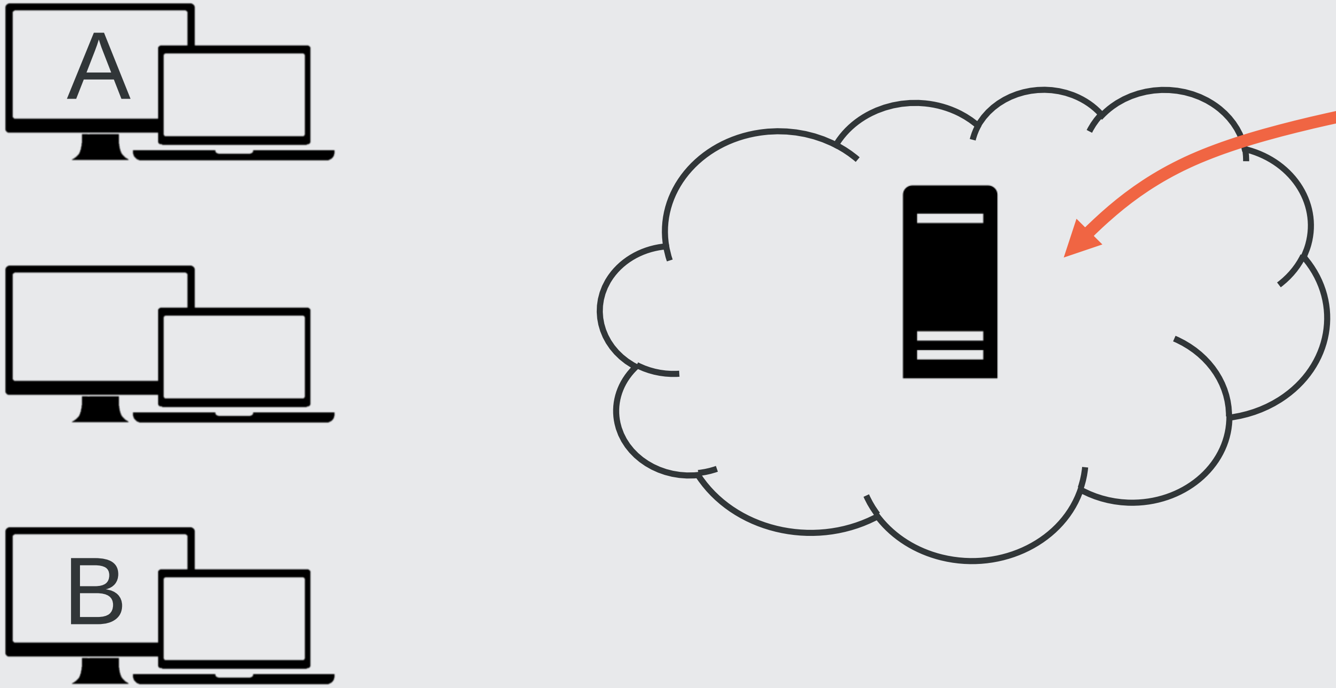
Group communication

- Processes >100K messages/min with 4 servers (scales linearly with # servers)

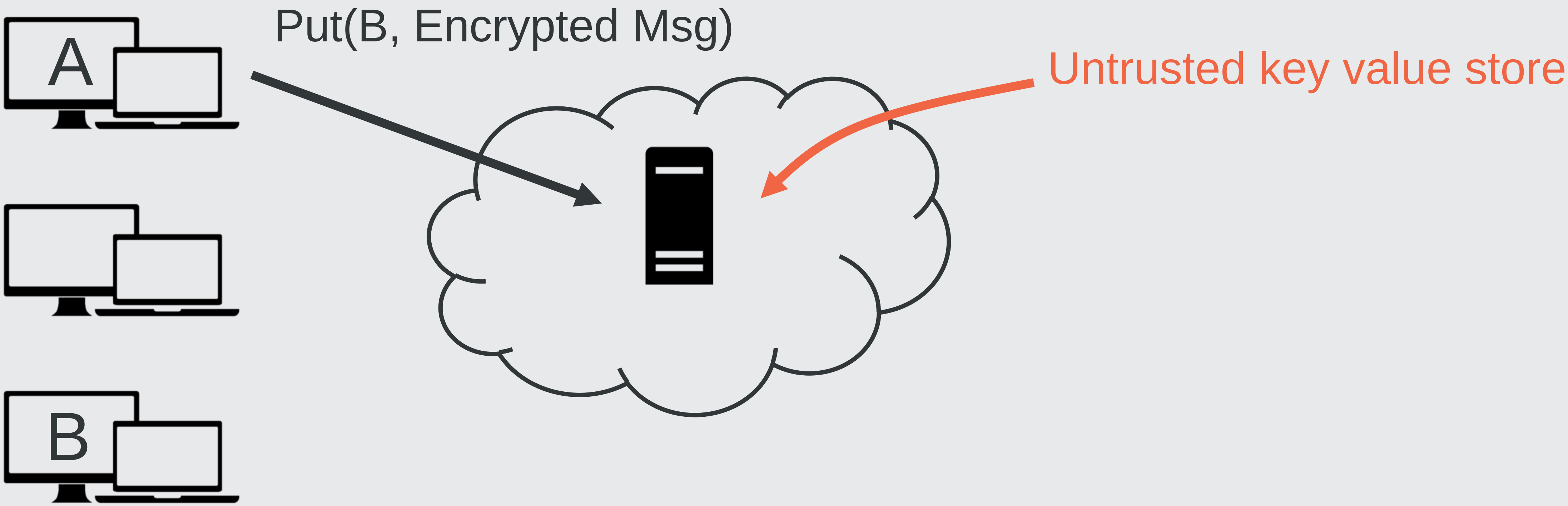
In the rest of this talk we answer

- How does Pung work?
- What is the performance of Pung?

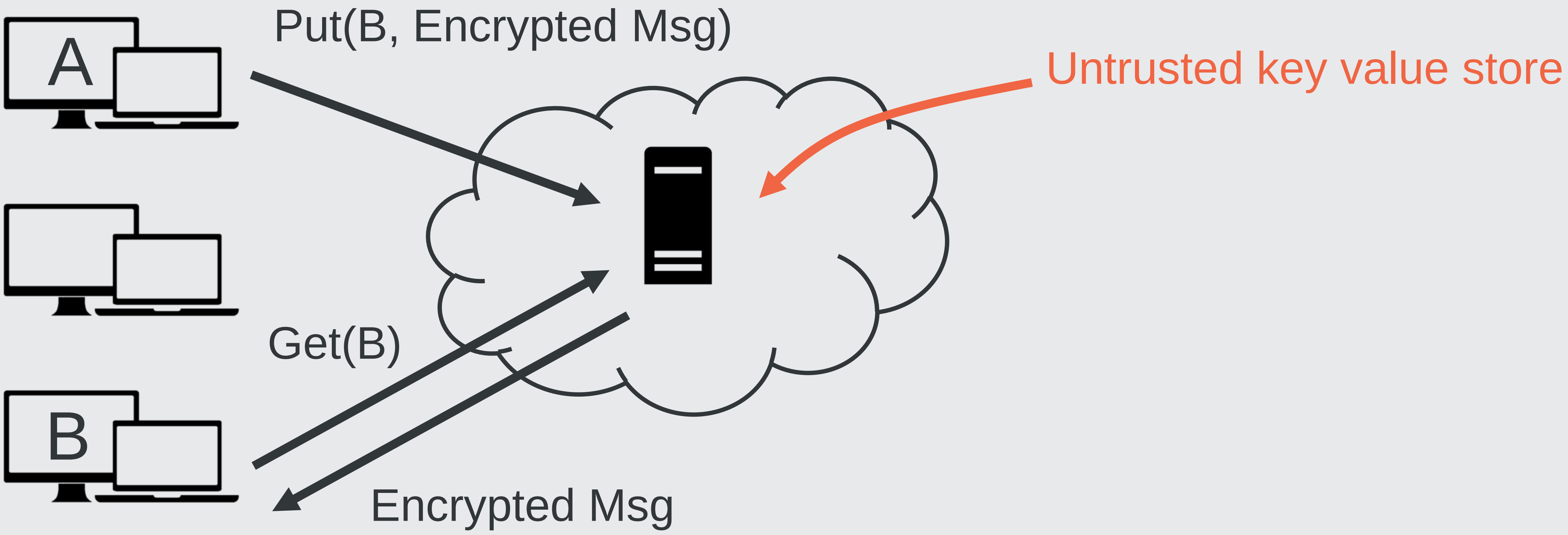
Clients use a key value store to communicate



Clients use a key value store to communicate



Clients use a key value store to communicate



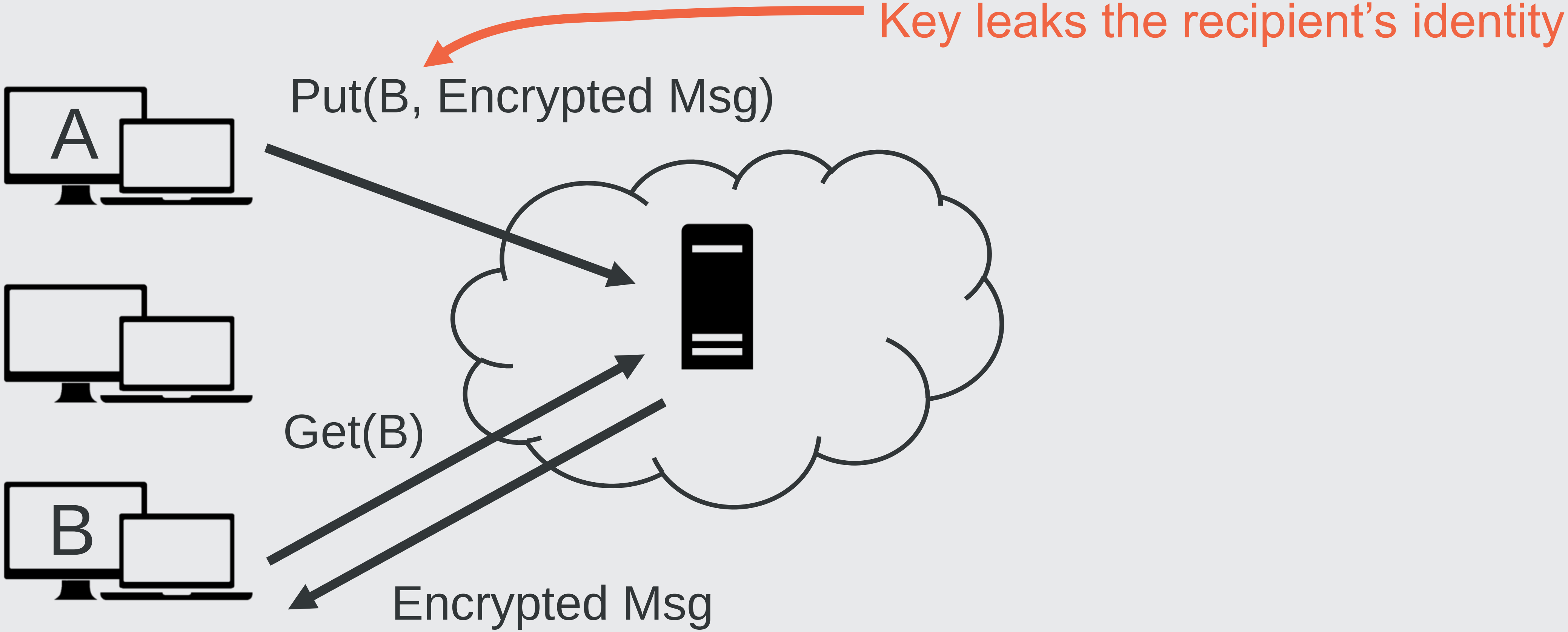
Pung must hide a lot of metadata

- Participants of a conversation
- Message size
- Time of a message being sent
- Time of message delivery
- Frequency of communication

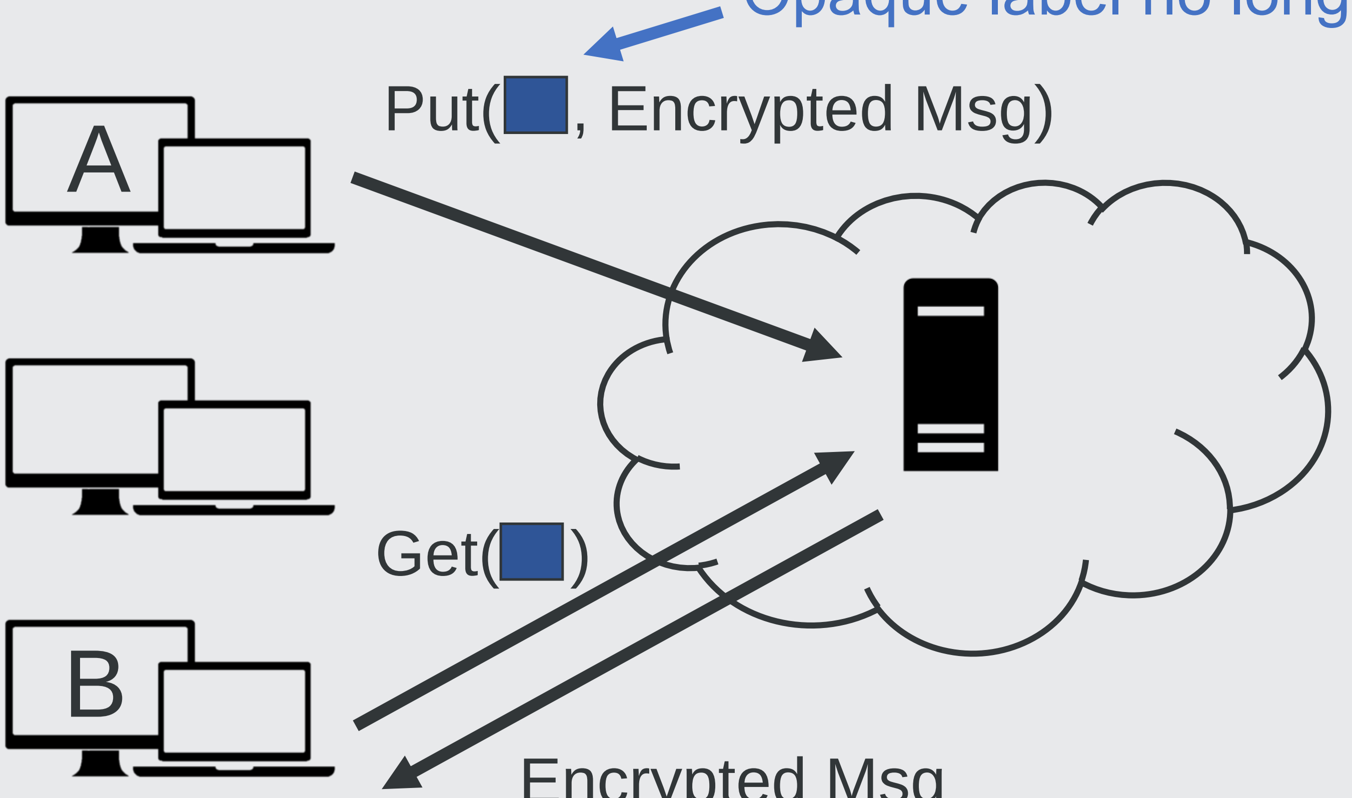
Pung must hide a lot of metadata

- Participants of a conversation
- Message size
- Time of a message being sent
- Time of message delivery
- Frequency of communication

Put request parameter leaks recipient



Put request parameter leaks recipient



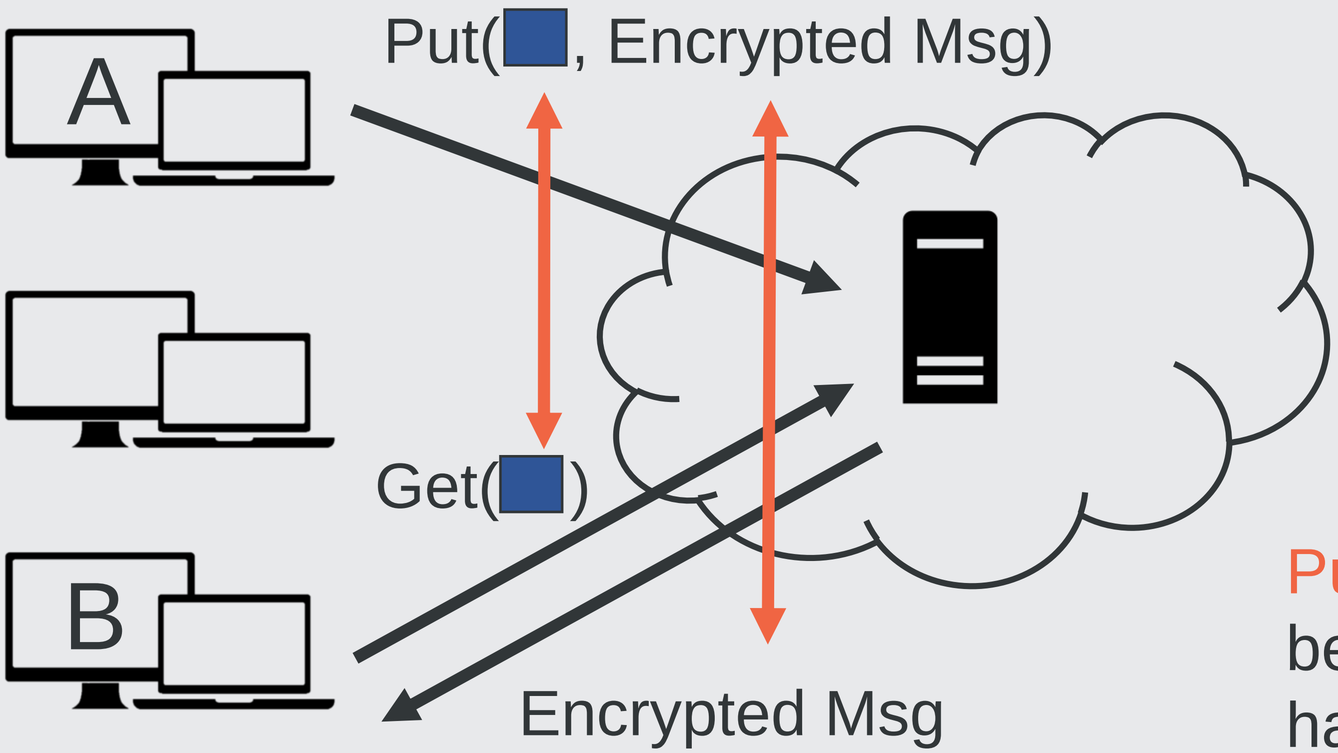
Opaque label no longer leak recipient’s identity!

Put(🟦, Encrypted Msg)

Get(🟦)

Encrypted Msg

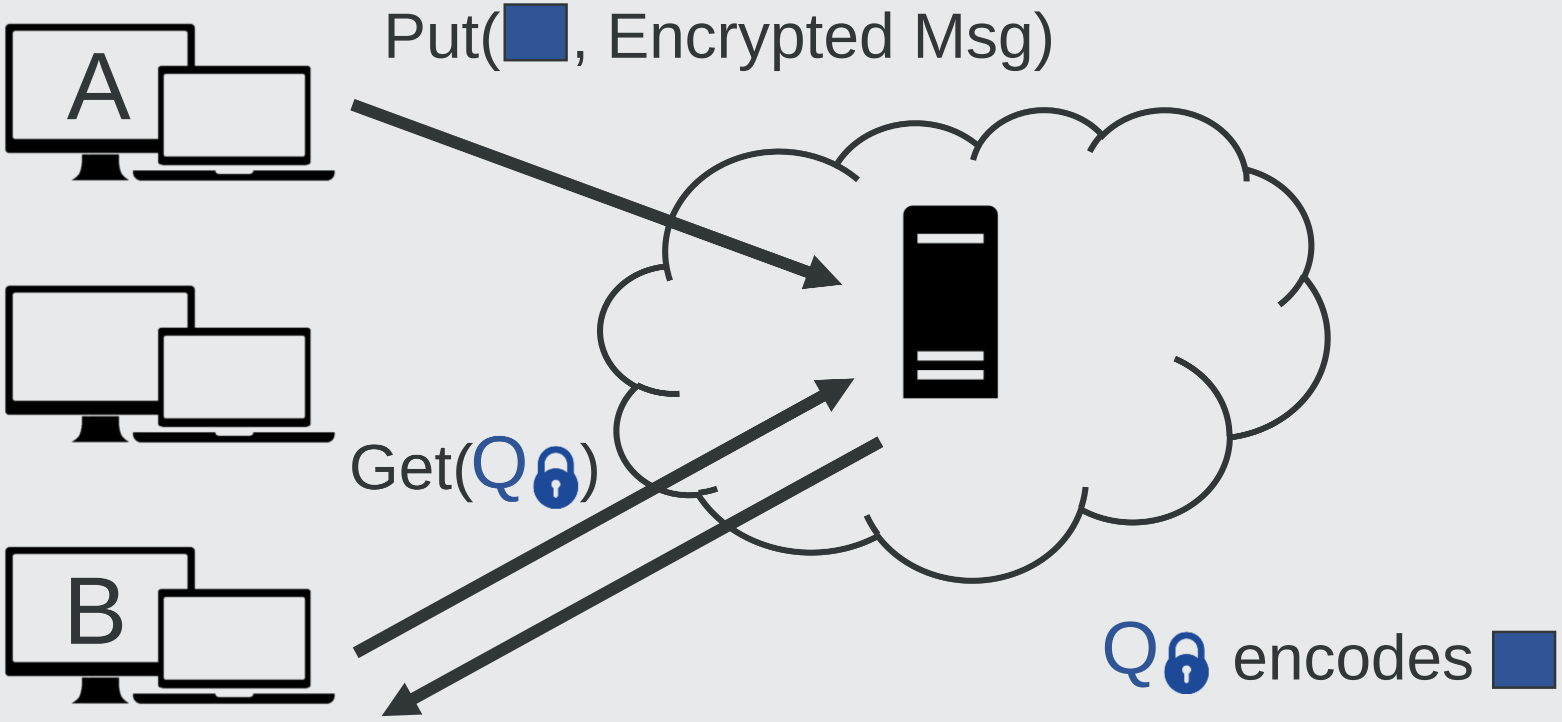
Put + Get in combination leak metadata!



Put from A and Get from B can be associated because they have the same inputs/outputs

➔ A is talking to B

Solution: break association of Put and Get

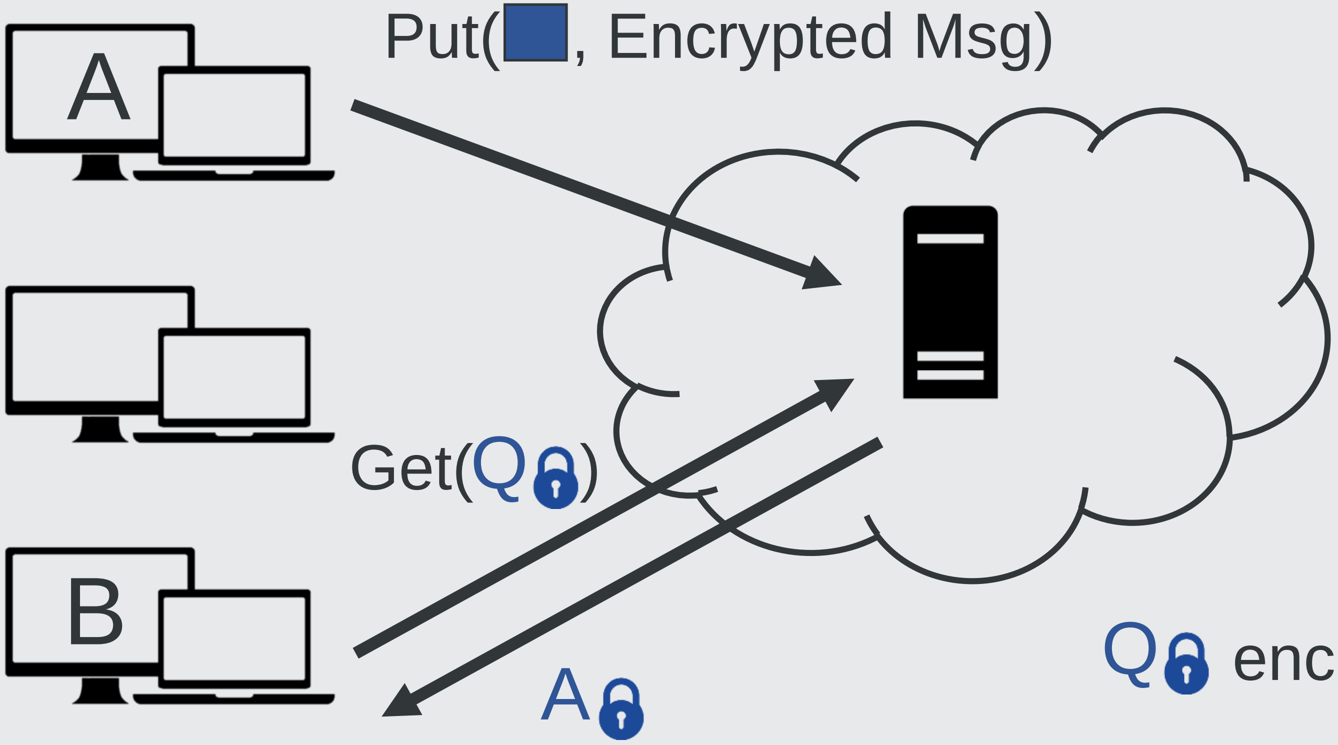


Put(🟦, Encrypted Msg)

Get(Q🔒)

Q🔒 encodes 🟦

Solution: break association of Put and Get



Put(🟦, Encrypted Msg)

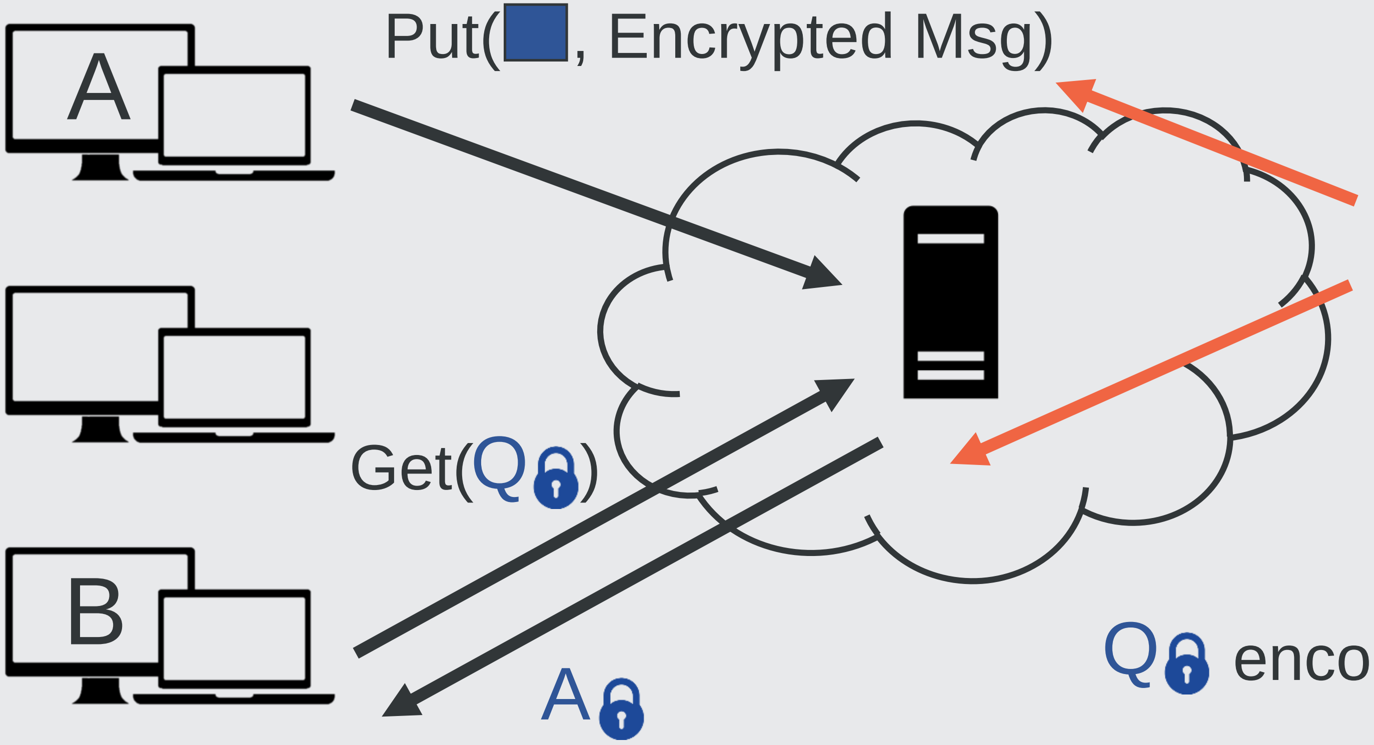
Get(Q🔒)

A🔒

Q🔒 encodes 🟦

A🔒 encodes “Encrypted Msg”

Solution: break association of Put and Get

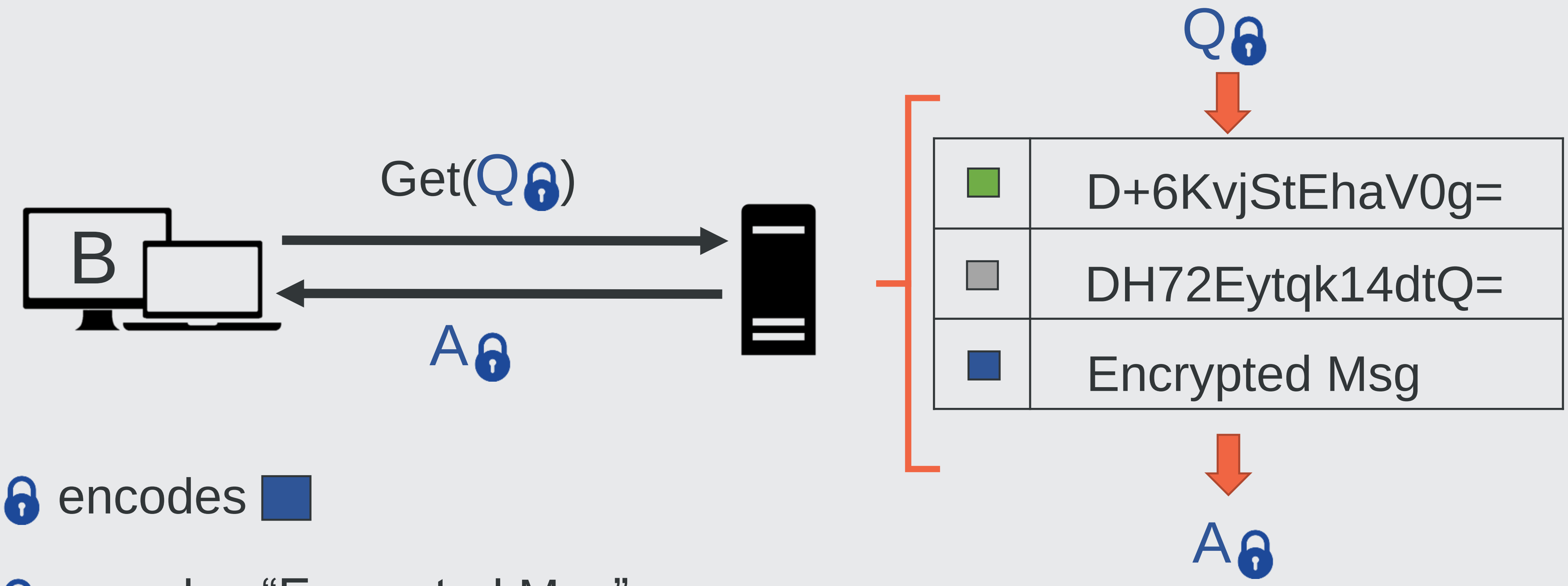


Put and Get cannot be associated since they don’t share anything distinguishable

Q🔒 encodes 🟦

A🔒 encodes “Encrypted Msg”

Server can answer the Query **obliviously**



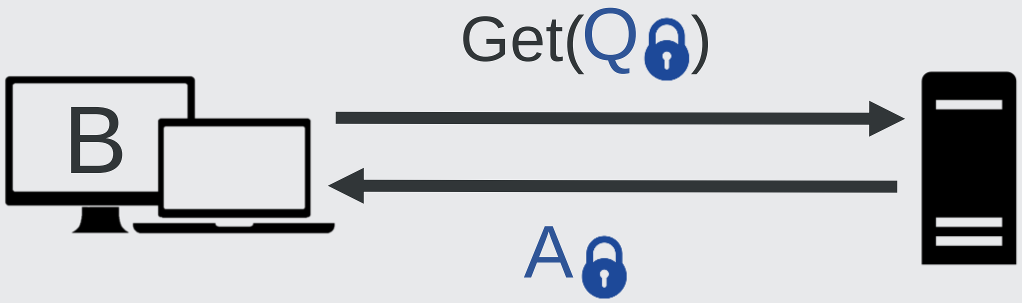
🟩	D+6KvjStEhaV0g=
⬜	DH72Eytqk14dtQ=
🟦	Encrypted Msg

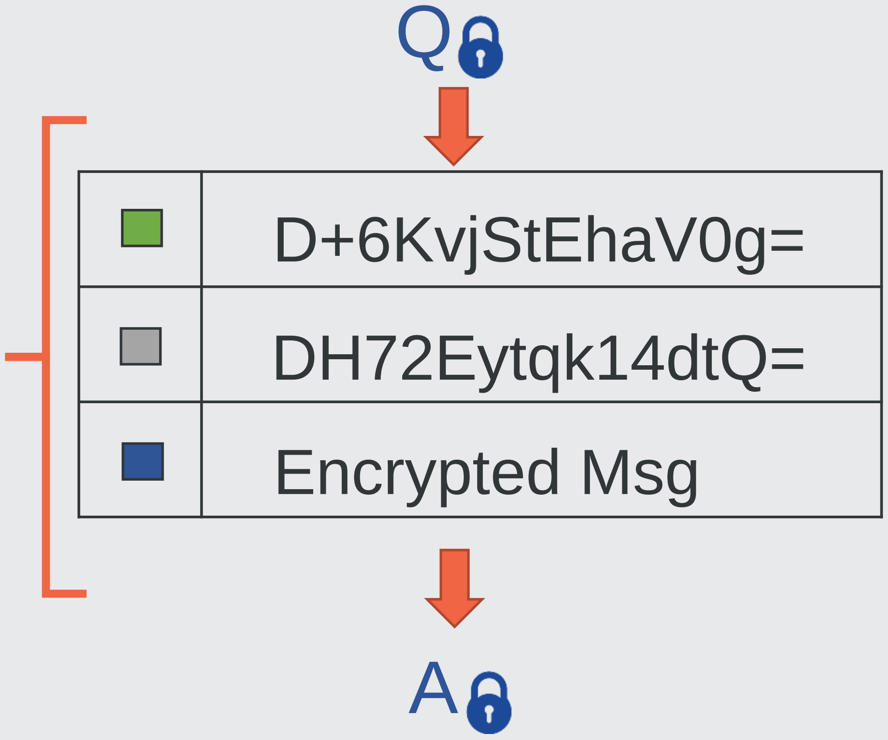
Q🔒 encodes 🟦

A🔒 encodes “Encrypted Msg”

Server can answer the Query **obliviously**

Private information retrieval (PIR) hides the access pattern by requiring the server to perform cryptographic operations over every single entry





🟩	D+6KvjStEhaV0g=
⬜	DH72Eytqk14dtQ=
🟦	Encrypted Msg

Q🔒 encodes 🟦

A🔒 encodes “Encrypted Msg”

Many applications benefit from clients retrieving messages in a batch

slack

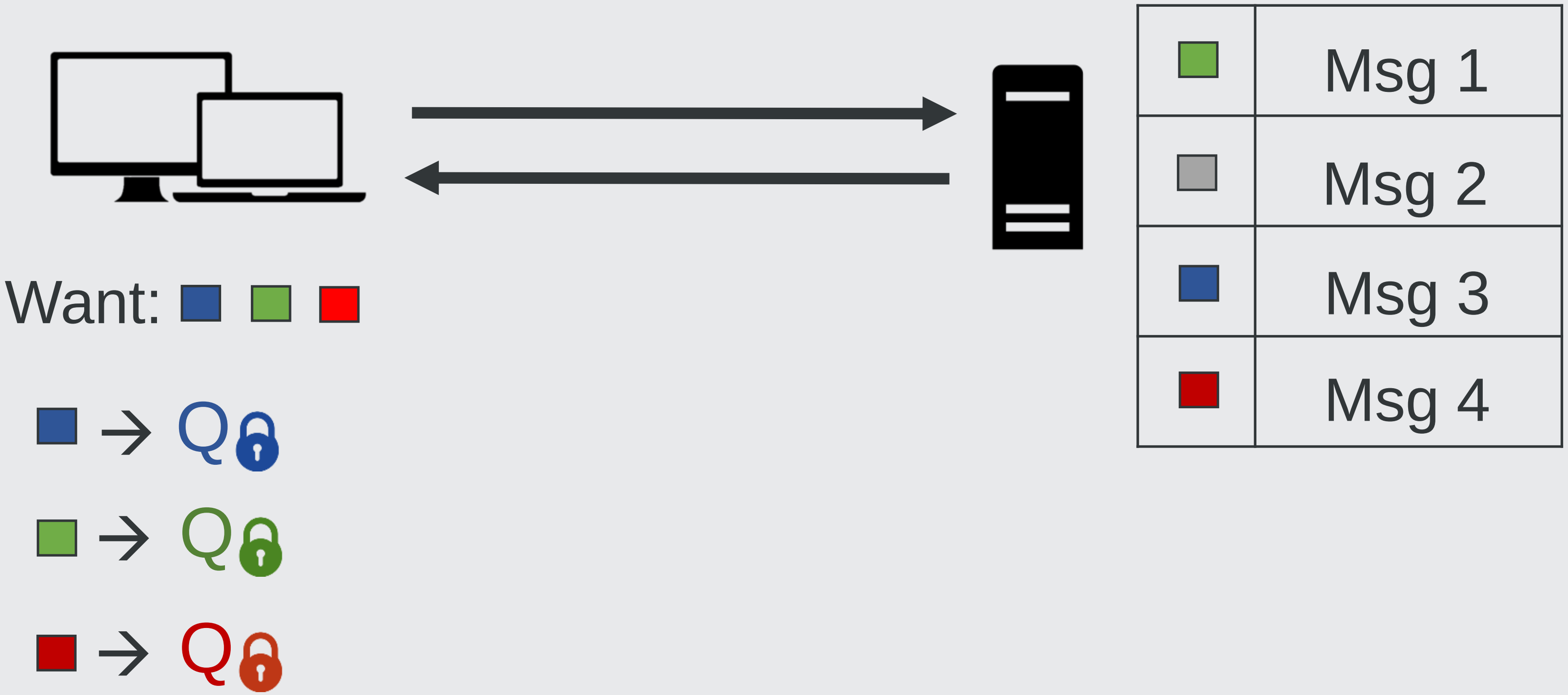
2,016

Clients can get k elements by issuing k queries

Want: 🟦 🟩 🟥

🟩	Msg 1
⬜	Msg 2
🟦	Msg 3
🟥	Msg 4

Clients can get k elements by issuing k queries



🟩	Msg 1
⬜	Msg 2
🟦	Msg 3
🟥	Msg 4

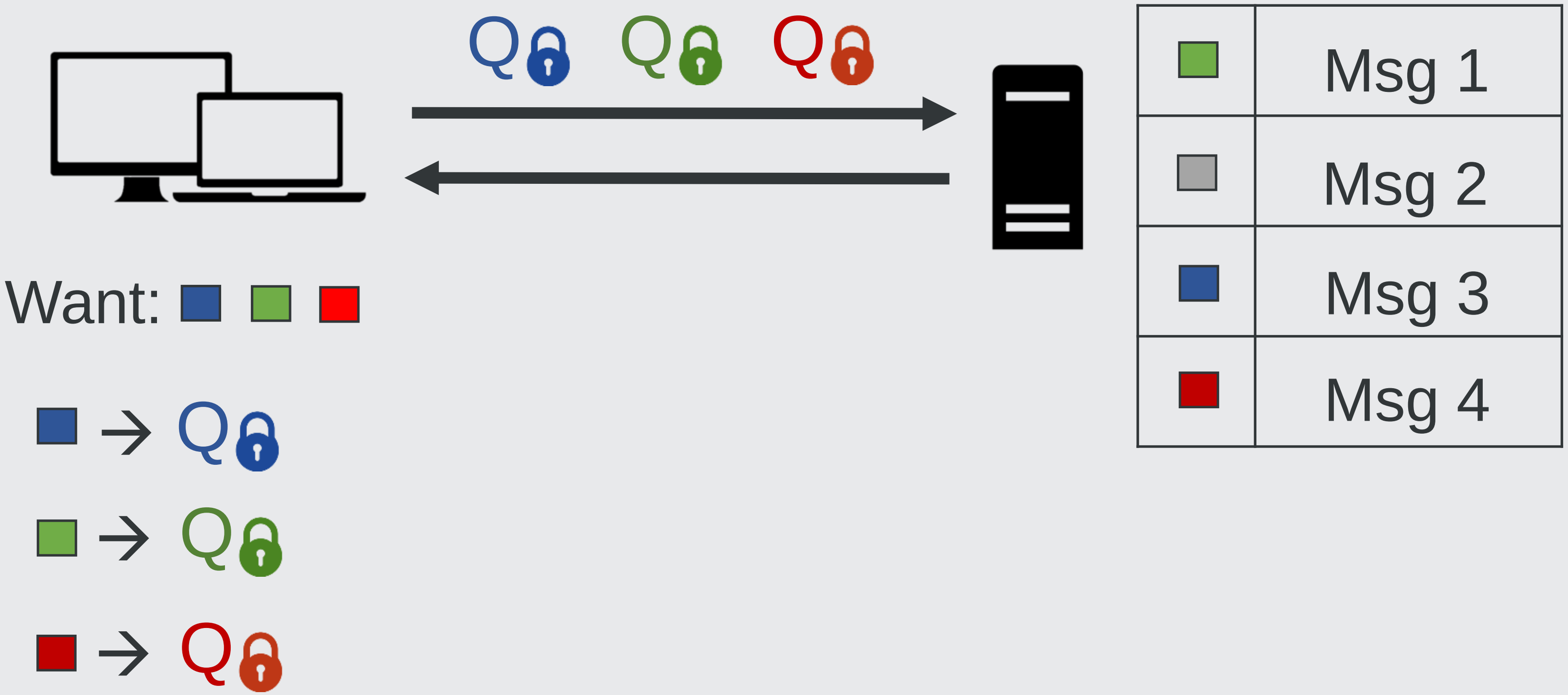
Want: 🟦 🟩 🟥

🟦 → Q🔒

🟩 → Q🔒

🟥 → Q🔒

Clients can get k elements by issuing k queries



Want: 🟦 🟩 🟥

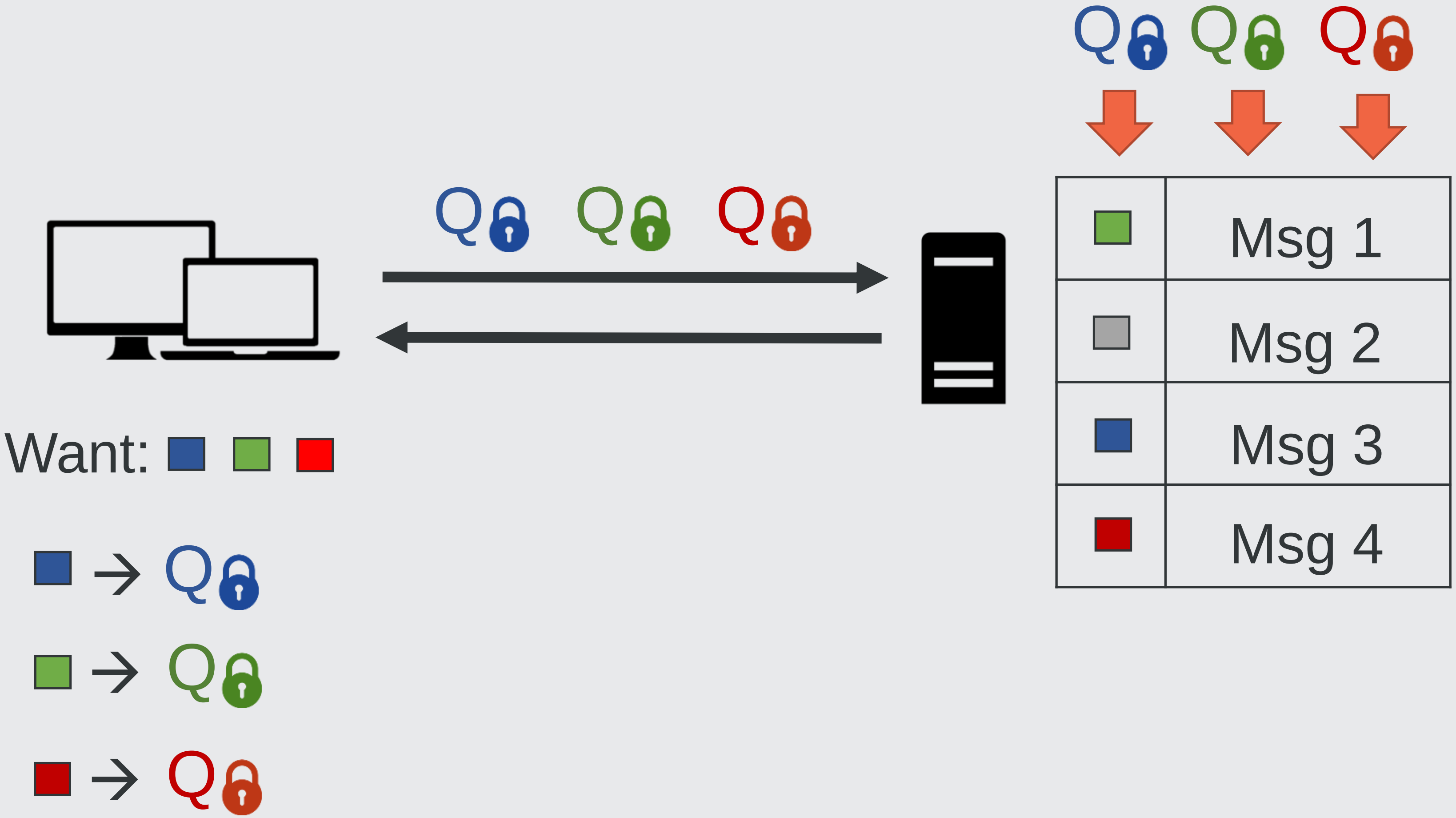
🟦 → Q🔒

🟩 → Q🔒

🟥 → Q🔒

🟩	Msg 1
⬜	Msg 2
🟦	Msg 3
🟥	Msg 4

Clients can get k elements by issuing k queries



Want: 🟦 🟩 🟥

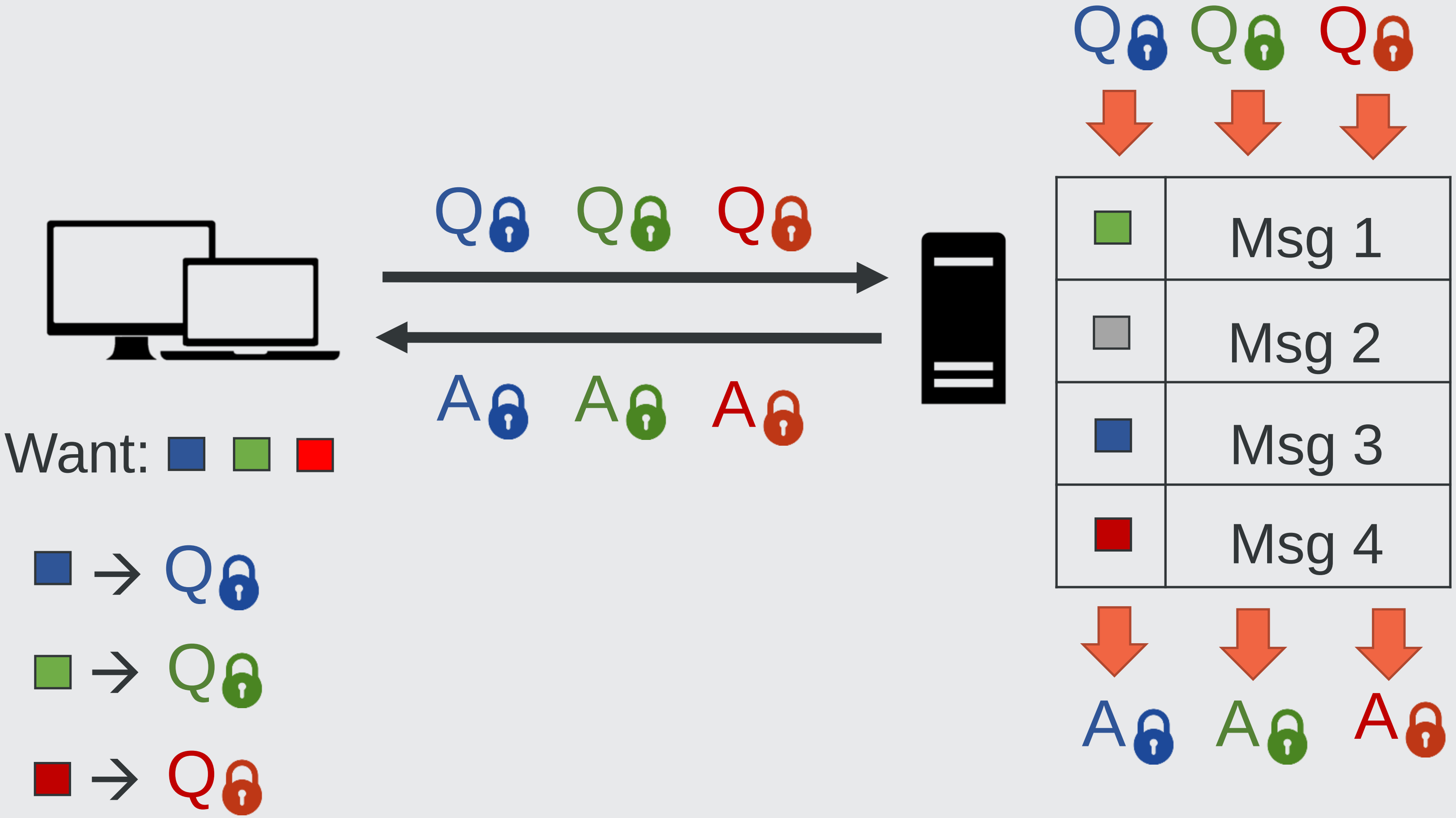
🟦 → Q

🟩 → Q

🟥 → Q

🟩	Msg 1
⬜	Msg 2
🟦	Msg 3
🟥	Msg 4

Clients can get k elements by issuing k queries



🟩	Msg 1
⬜	Msg 2
🟦	Msg 3
🟥	Msg 4

Want: 🟦 🟩 🟥

🟦 → Q

🟩 → Q

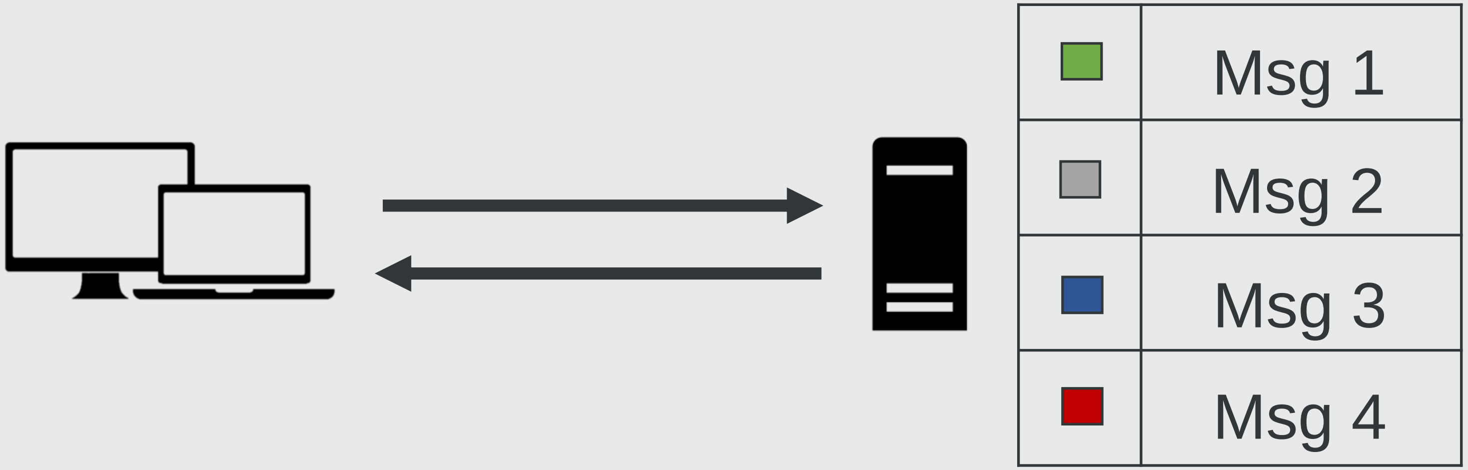
🟥 → Q

Server processes each query independently

Elements processed: kn = 12 (4 per query)

Can we amortize the cost of answering k Get requests?

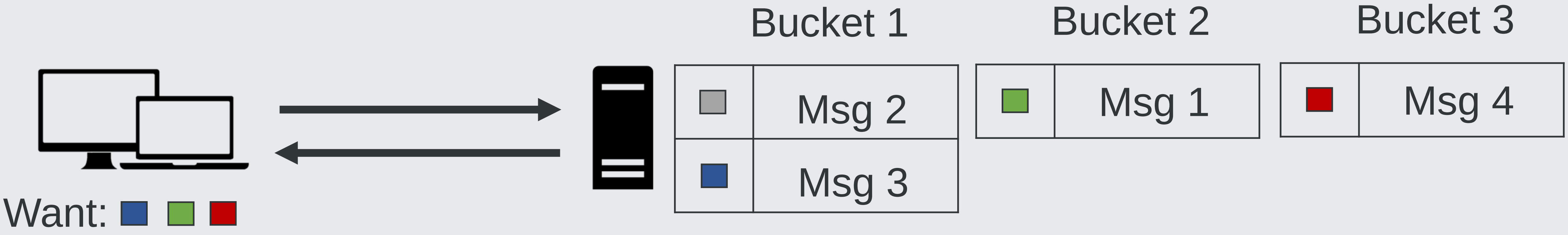
Idea 1: Partition the database into k buckets



🟩	Msg 1
⬜	Msg 2
🟦	Msg 3
🟥	Msg 4

Split database into k buckets with a static partitioning scheme

Idea 1: Partition the database into k buckets



Want: 🟦 🟩 🟥

Bucket 1

⬜	Msg 2
🟦	Msg 3

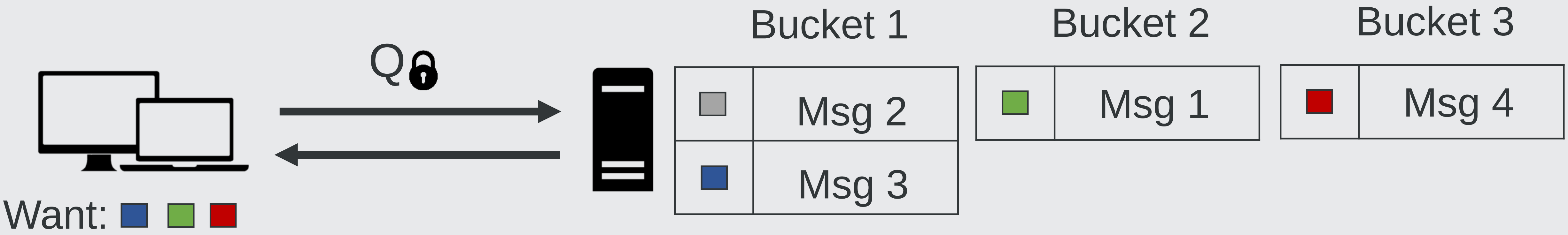
Bucket 2

🟩	Msg 1

Bucket 3

🟥	Msg 4

Idea 1: Partition the database into k buckets



Q🔒

Want: 🟦 🟩 🟥

Bucket 1

⬜	Msg 2
🟦	Msg 3

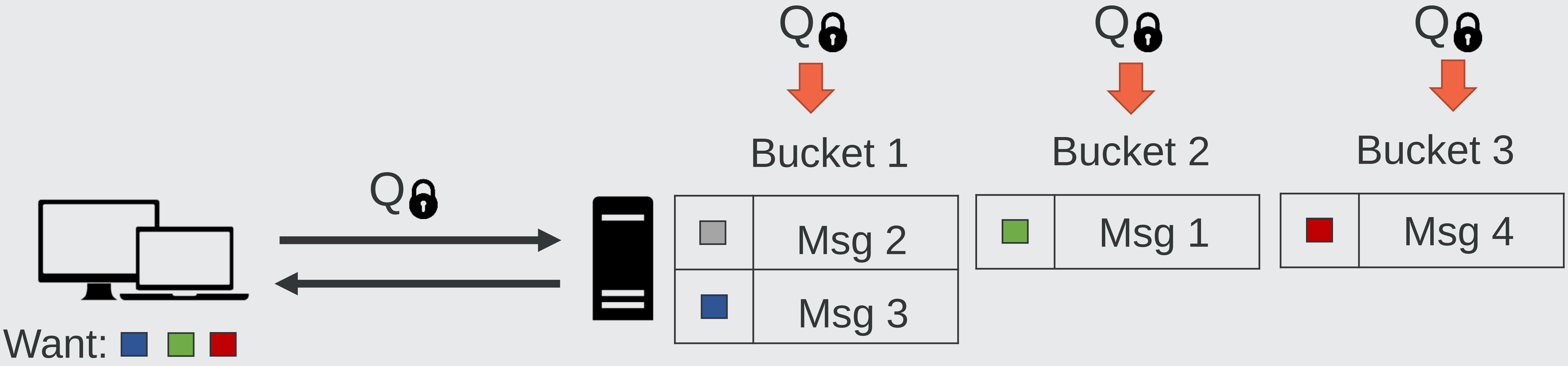
Bucket 2

🟩	Msg 1

Bucket 3

🟥	Msg 4

Idea 1: Partition the database into k buckets



Q🔒

Want: 🟦 🟩 🟥

Q🔒 Q🔒 Q🔒

Bucket 1

⬜	Msg 2
🟦	Msg 3

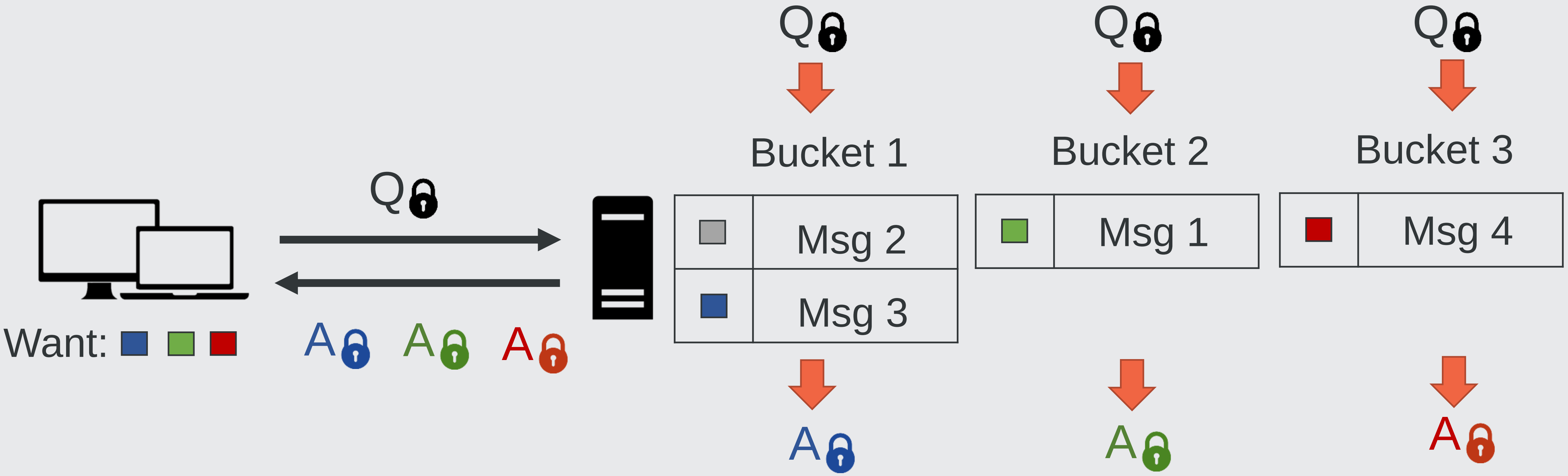
Bucket 2

🟩	Msg 1

Bucket 3

🟥	Msg 4

Idea 1: Partition the database into k buckets



Q🔒

Want: 🟦 🟩 🟥

A🔒 A🔒 A🔒

Q🔒 Q🔒 Q🔒

Bucket 1

⬜ Msg 2

🟦 Msg 3

A🔒

Bucket 2

🟩 Msg 1

A🔒

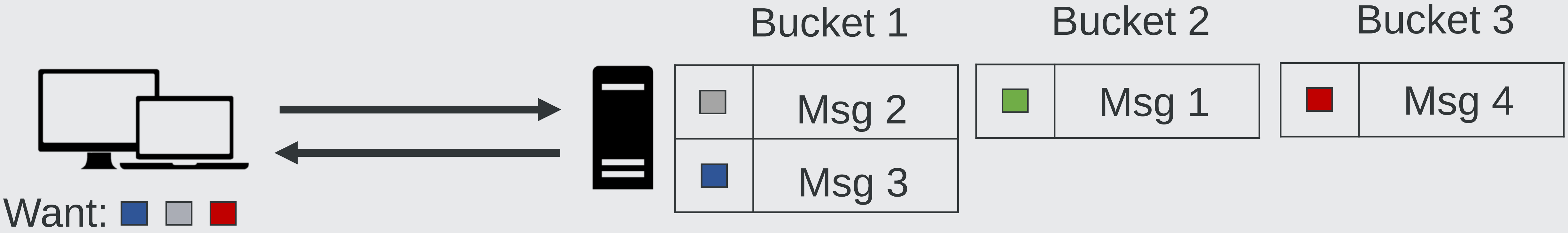
Bucket 3

🟥 Msg 4

A🔒

Elements processed: n = 4 (8 fewer than before)

Issue: how does a client get >1 message from the same bucket?



Bucket 1

◻ Msg 2

🟦 Msg 3

Bucket 2

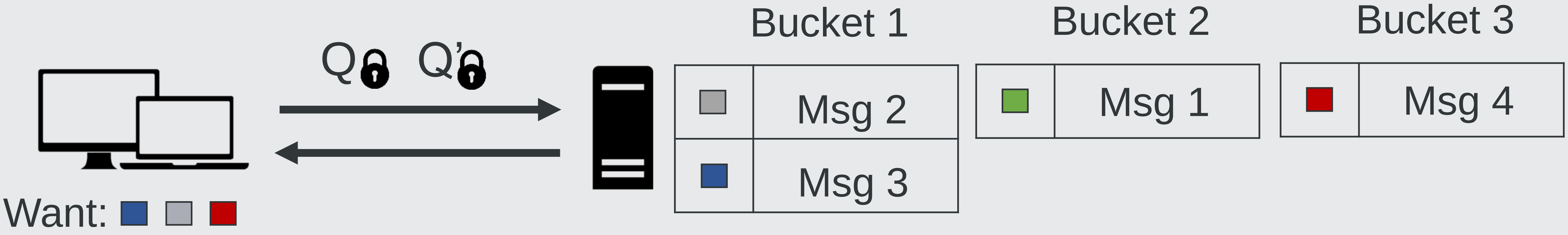
🟩 Msg 1

Bucket 3

🟥 Msg 4

Want: 🟦 ◻ 🟥

Issue: how does a client get >1 message from the same bucket?



Q Q'

Bucket 1

⬜ Msg 2

🟦 Msg 3

Bucket 2

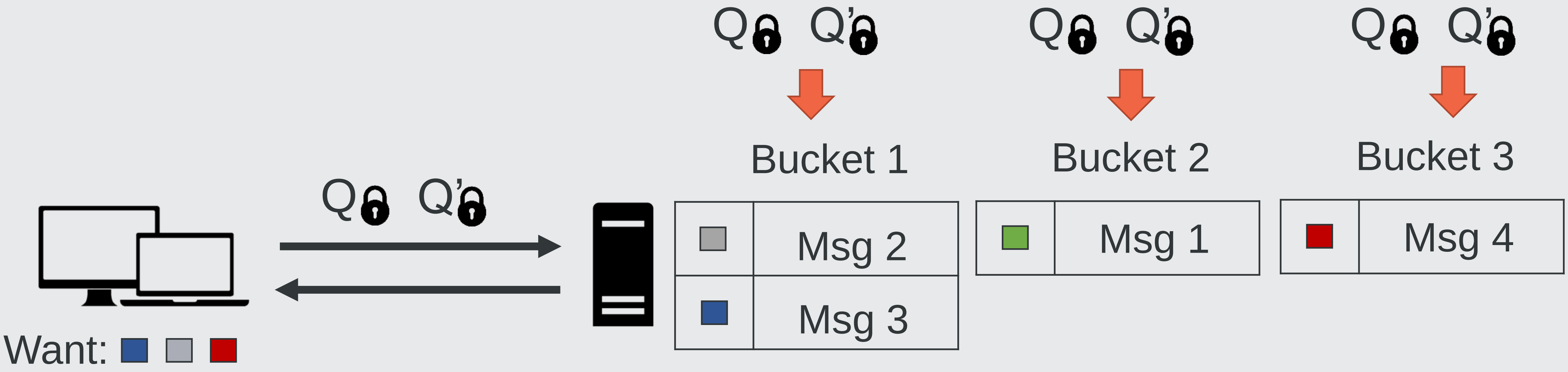
🟩 Msg 1

Bucket 3

🟥 Msg 4

Want: 🟦 ⬜ 🟥

Issue: how does a client get >1 message from the same bucket?



Q🔒 Q'🔒

Want: 🟦 ⬜ 🟥

Q🔒 Q'🔒

Bucket 1

⬜	Msg 2
🟦	Msg 3

Q🔒 Q'🔒

Bucket 2

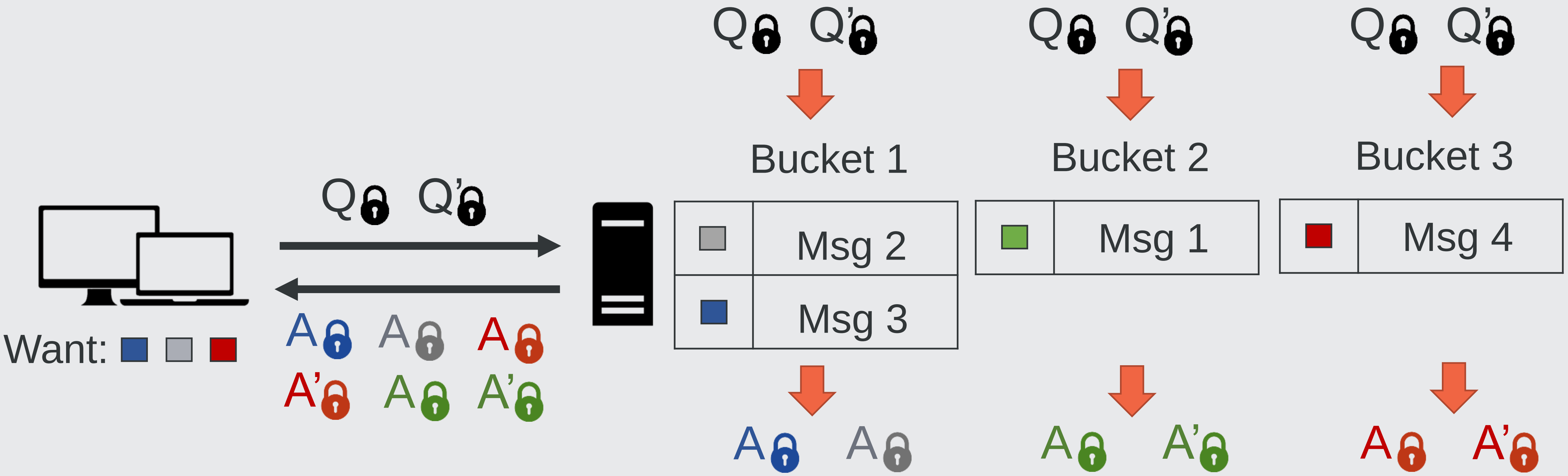
🟩	Msg 1

Q🔒 Q'🔒

Bucket 3

🟥	Msg 4

Issue: how does a client get >1 message from the same bucket?



Q Q'

Q Q' Q Q' Q Q'

Bucket 1 Bucket 2 Bucket 3

⬜	Msg 2
🟦	Msg 3

🟩	Msg 1

🟥	Msg 4

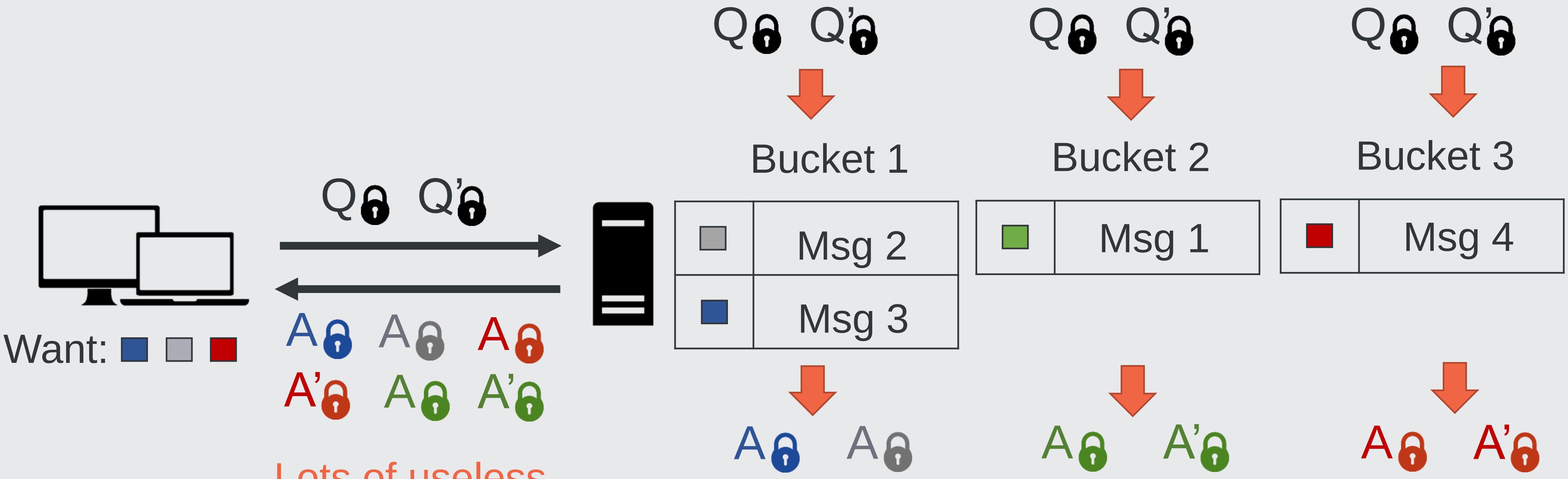
Want: 🟦 ⬜ 🟥

A A A
A' A A'

A A A A' A A'

Elements processed: 8 (4 fewer than before)

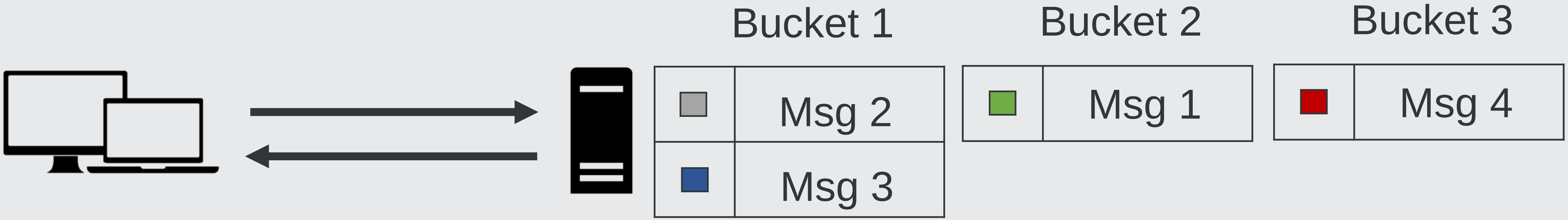
Issue: how does a client get >1 message from the same bucket?



Lots of useless answers (overhead)

Elements processed: 8 (4 fewer than before)

Idea 2: Alias messages under two labels



Bucket 1

⬜	Msg 2
🟦	Msg 3

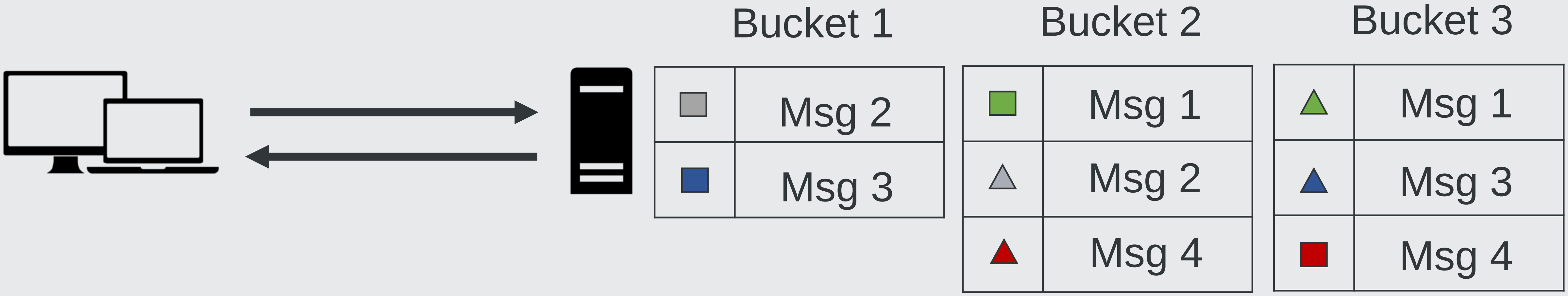
Bucket 2

🟩	Msg 1

Bucket 3

🟥	Msg 4

Idea 2: Alias messages under two labels



Bucket 1

⬜	Msg 2
🟦	Msg 3

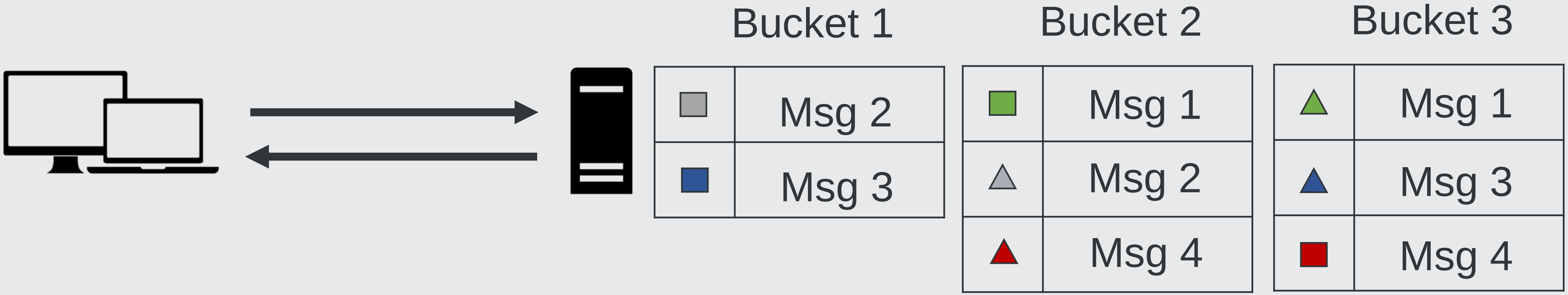
Bucket 2

🟩	Msg 1
△	Msg 2
🔺	Msg 4

Bucket 3

▲	Msg 1
▲	Msg 3
🟥	Msg 4

Idea 2: Alias messages under two labels



Bucket 1

◼	Msg 2
🟦	Msg 3

Bucket 2

🟩	Msg 1
△	Msg 2
🔺	Msg 4

Bucket 3

▲	Msg 1
▲	Msg 3
🟥	Msg 4

Any message can be found in 2 different buckets
→ doubles the cost of processing each query

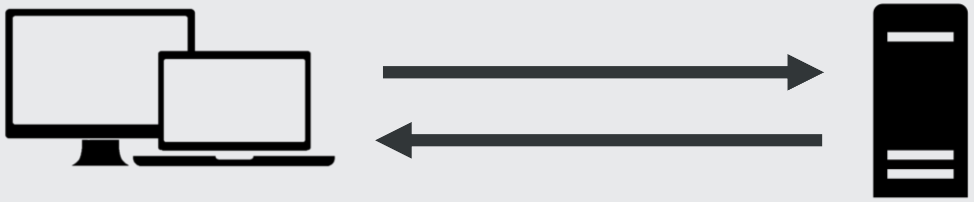
With aliasing, clients have multiple buckets from which to get a message

→ Clients can leverage the power of 2 choices

[Azar, Broder, Karlin, and Upfal, STOC ’94]

[Mitzenmacher, Ph.D. Thesis ‘96]

Idea 2: Alias messages under two labels



Want: 🟦 ⬜ 🟥

Table 1

⬜	Msg 2
🟦	Msg 3

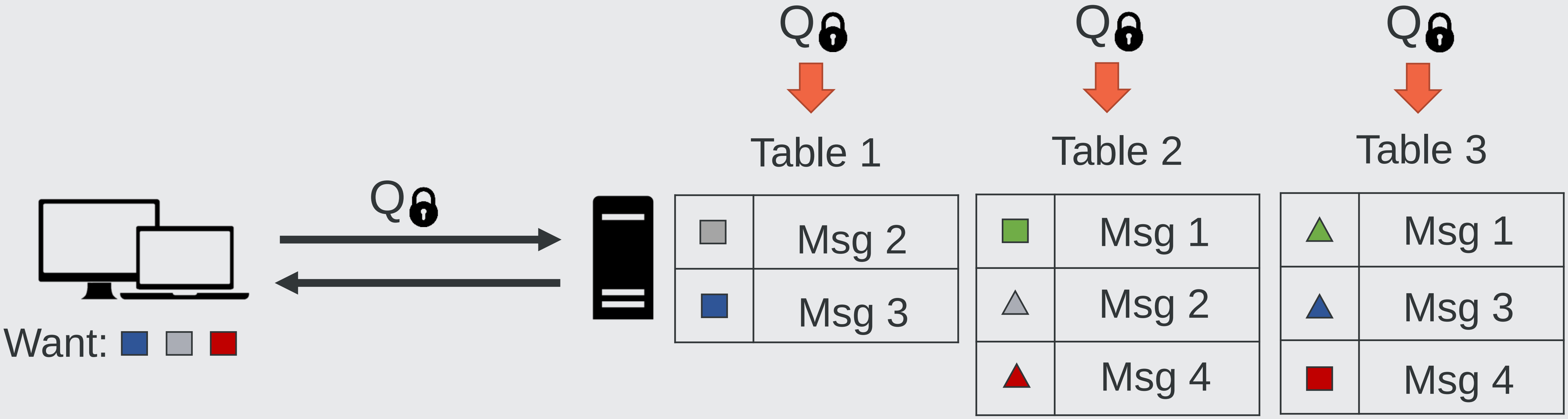
Table 2

🟩	Msg 1
△	Msg 2
🔺	Msg 4

Table 3

▲	Msg 1
▲	Msg 3
🟥	Msg 4

Idea 2: Alias messages under two labels



Want: 🟦 ◻ 🟥

Table 1

◻	Msg 2
🟦	Msg 3

Table 2

🟩	Msg 1
△	Msg 2
▲	Msg 4

Table 3

▲	Msg 1
▲	Msg 3
🟥	Msg 4

Idea 2: Alias messages under two labels

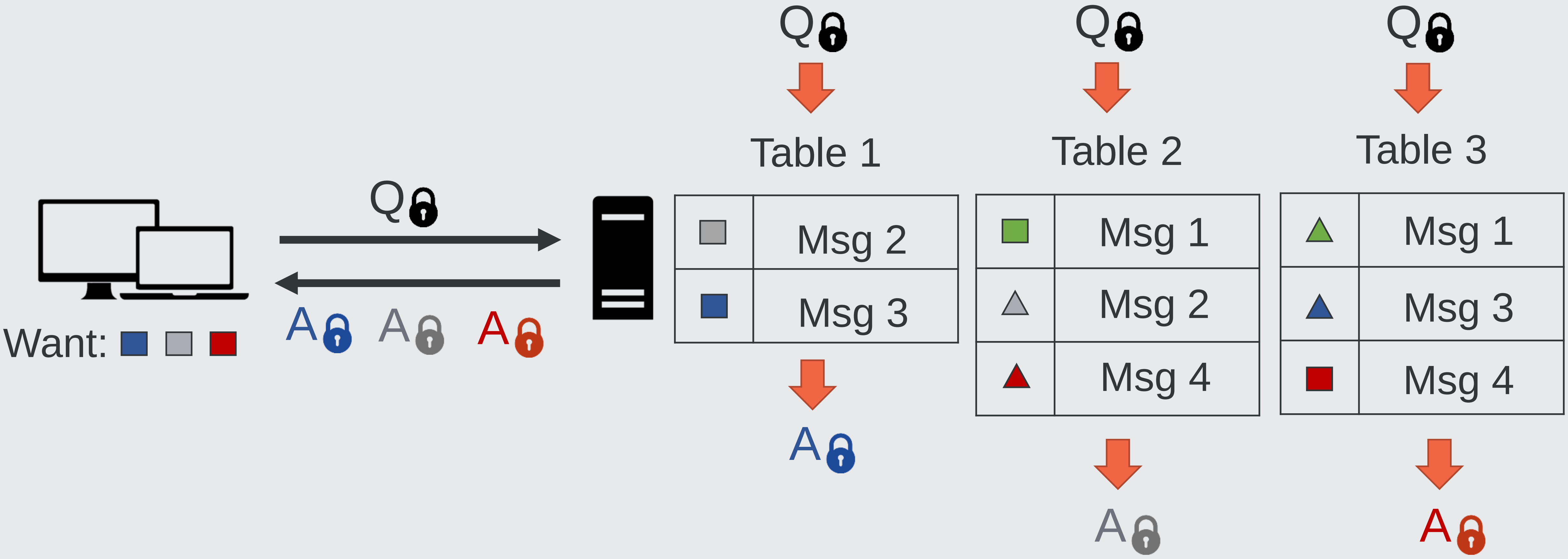


Table 1

◼	Msg 2
🟦	Msg 3

Table 2

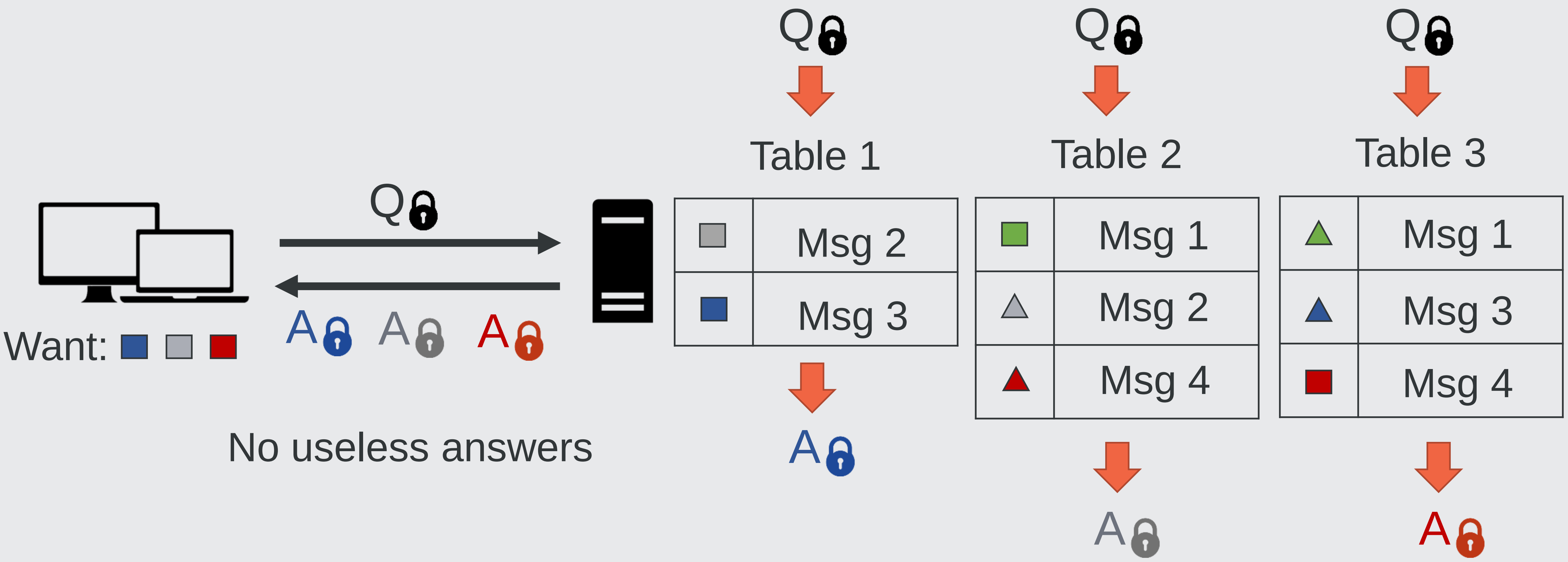
🟩	Msg 1
▲	Msg 2
▲	Msg 4

Table 3

▲	Msg 1
▲	Msg 3
🟥	Msg 4

Elements processed: 8 (4 fewer than before)

Idea 2: Alias messages under two labels



Want: 🟦 ◼ 🟥

Table 1

◼	Msg 2
🟦	Msg 3

Table 2

🟩	Msg 1
△	Msg 2
🔺	Msg 4

Table 3

▲	Msg 1
▲	Msg 3
🟥	Msg 4

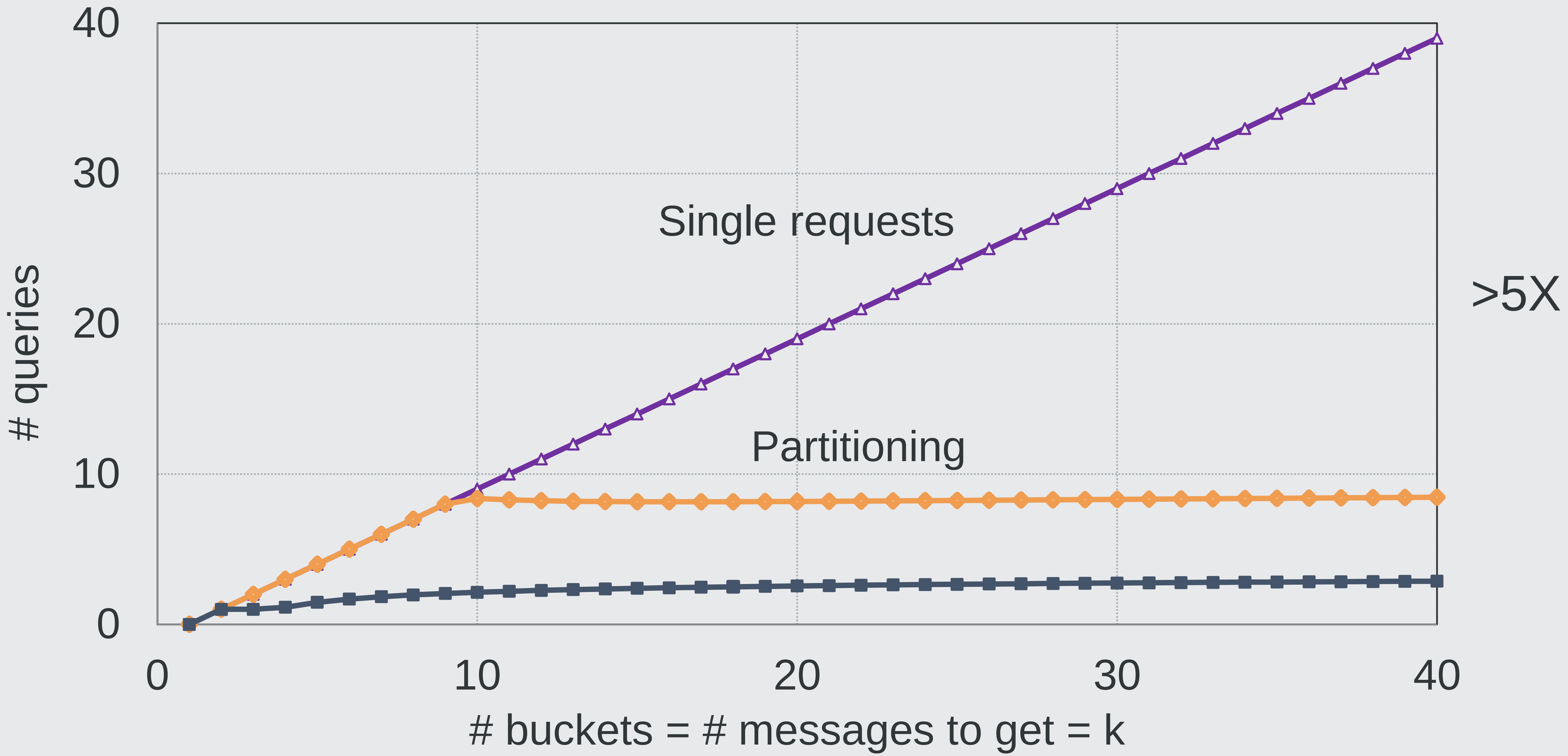
No useless answers

Elements processed: 8 (4 fewer than before)

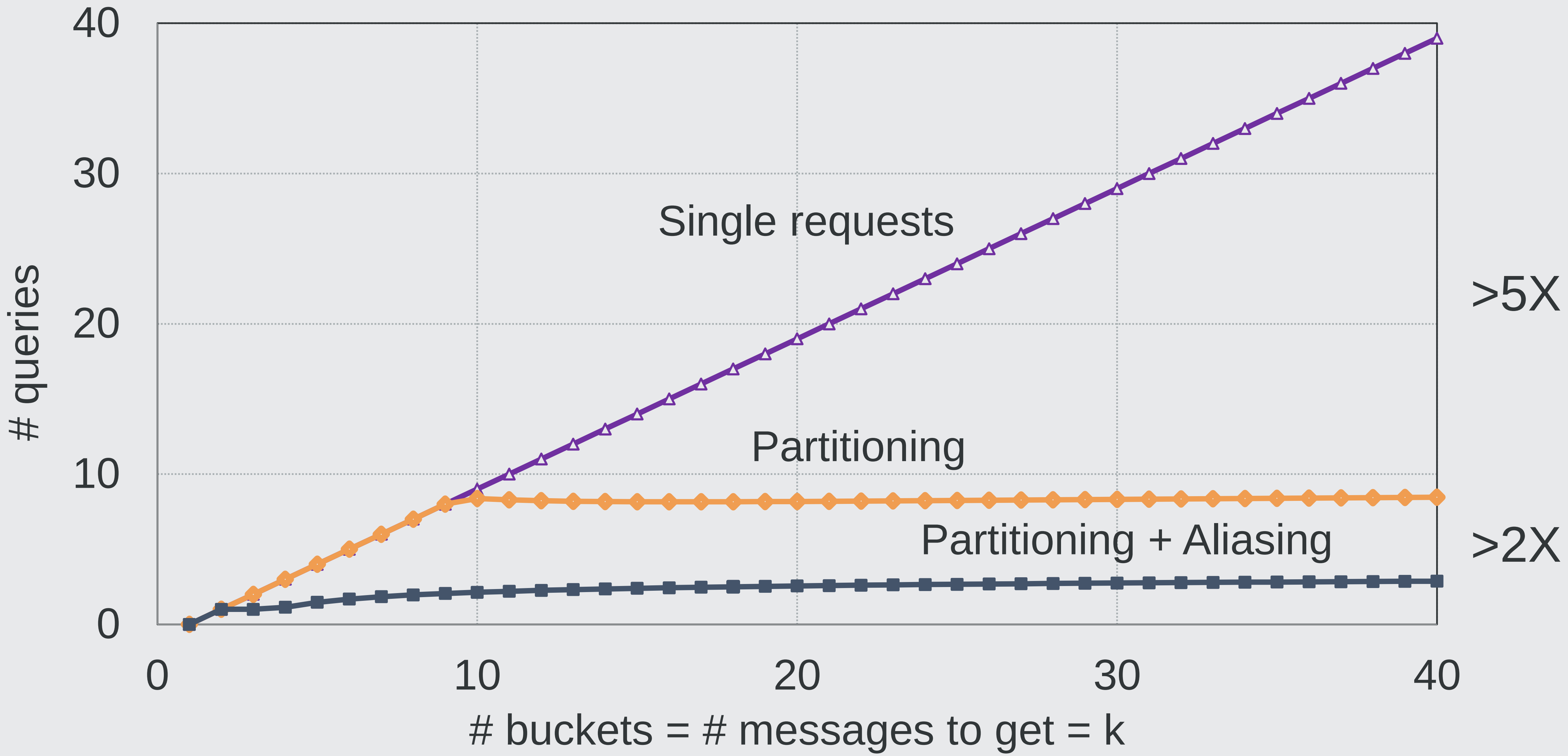
Queries required to get any k messages

Single requests

Queries required to get any k messages



Queries required to get any k messages



In the paper we also discuss

- How to encode buckets so that one query is sufficient
- How to construct queries if clients do not know the layout of the server’s database

In the rest of this talk we answer

- How does Pung work?
- What is the performance of Pung?

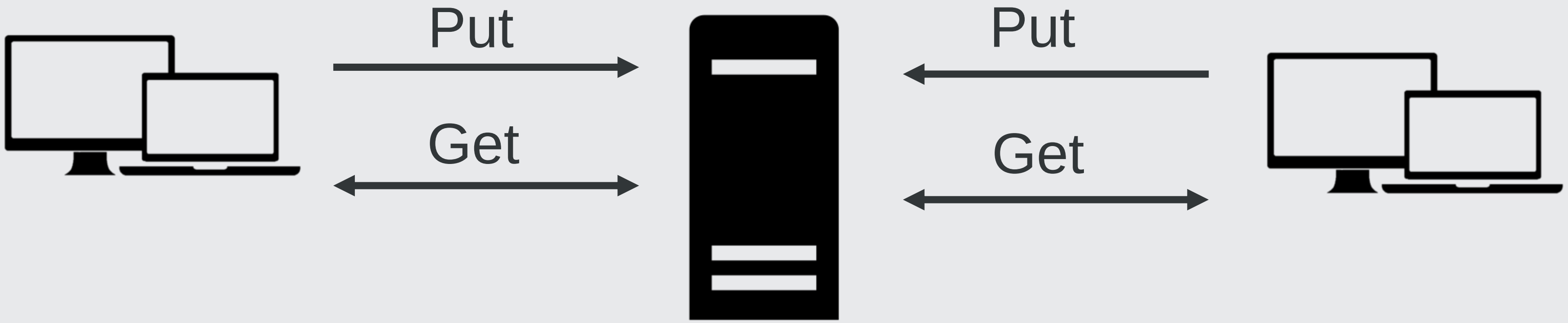
Pung’s prototype

- 5K source lines of Rust
- PIR library is XPIR [Aguilar-Melchor et al., PETS 2016]
- Pung’s server-side computation expressed as a dataflow graph
  - Runs on a Naiad cluster (using the timely dataflow library)

Evaluation questions

- How many users and messages can Pung support?
- What is the throughput of Pung when batching?

Evaluation setup



Put

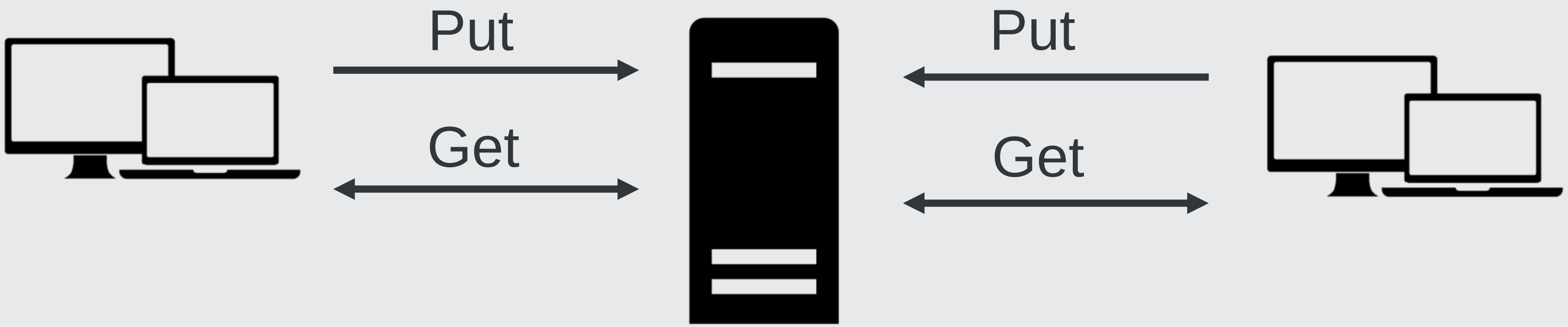
Get

Put

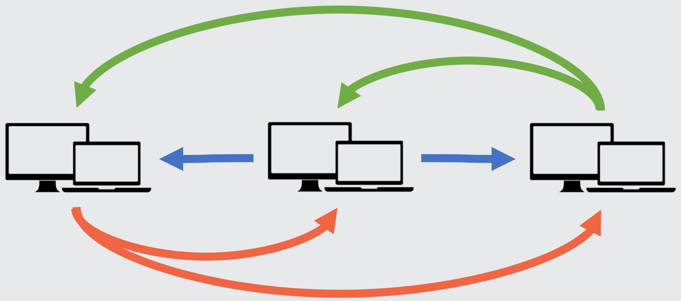
Get

Server is 64 dataflow workers across 4 VMs

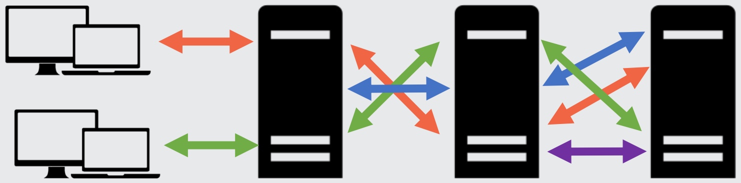
Evaluation setup



Server is 64 dataflow workers across 4 VMs



Dissent [CCS ‘10]



Vuvuzela [SOSP ’15]

How many users and messages can Pung support?

Number of users supported with 1 min latency

Dissent: ~64

1000X

Pung: ~65K

Vuvuzela: ~2M

Number of users supported with 1 min latency

Dissent: ~64

Pung: ~65K

1000X

Vuvuzela: ~2M

Dissent provides a stronger property than Pung and Vuvuzela

Number of users supported with 1 min latency

Dissent: ~64

Pung: ~65K

Vuvuzela: ~2M

1000X

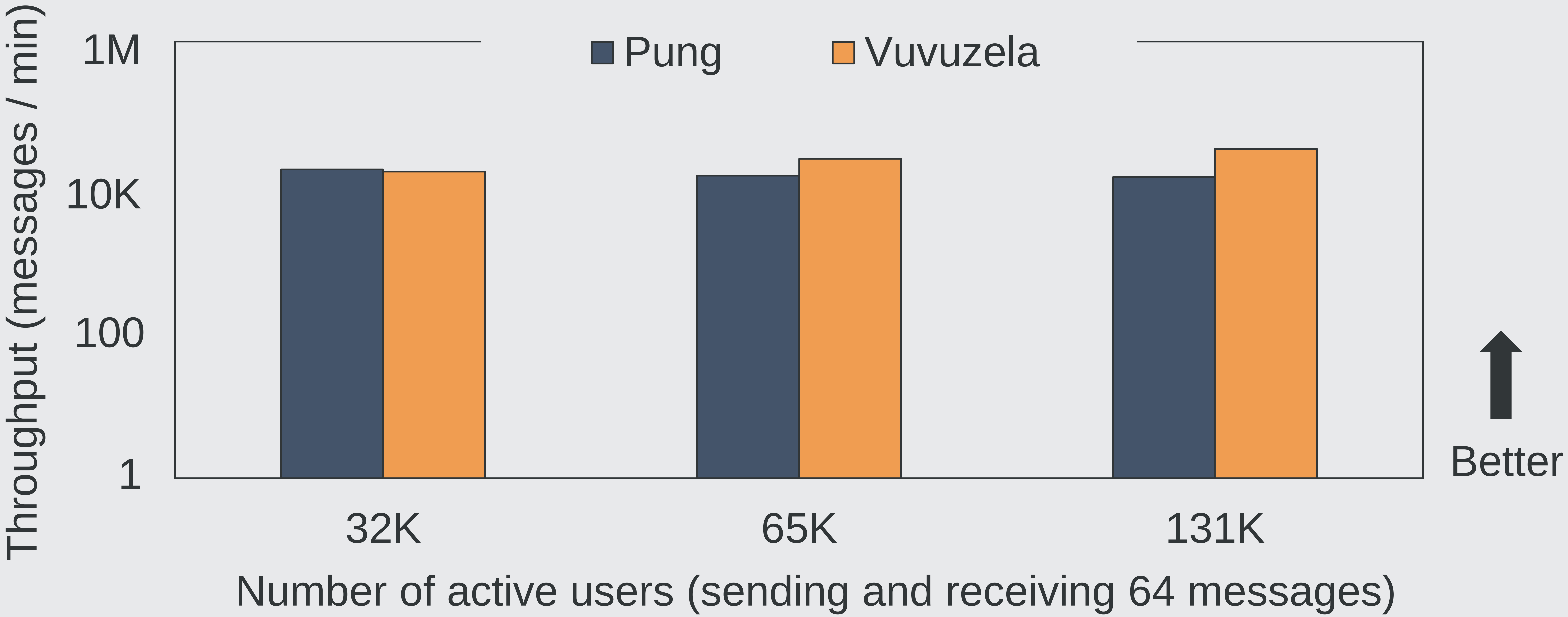
32X

Dissent provides a stronger property than Pung and Vuvuzela

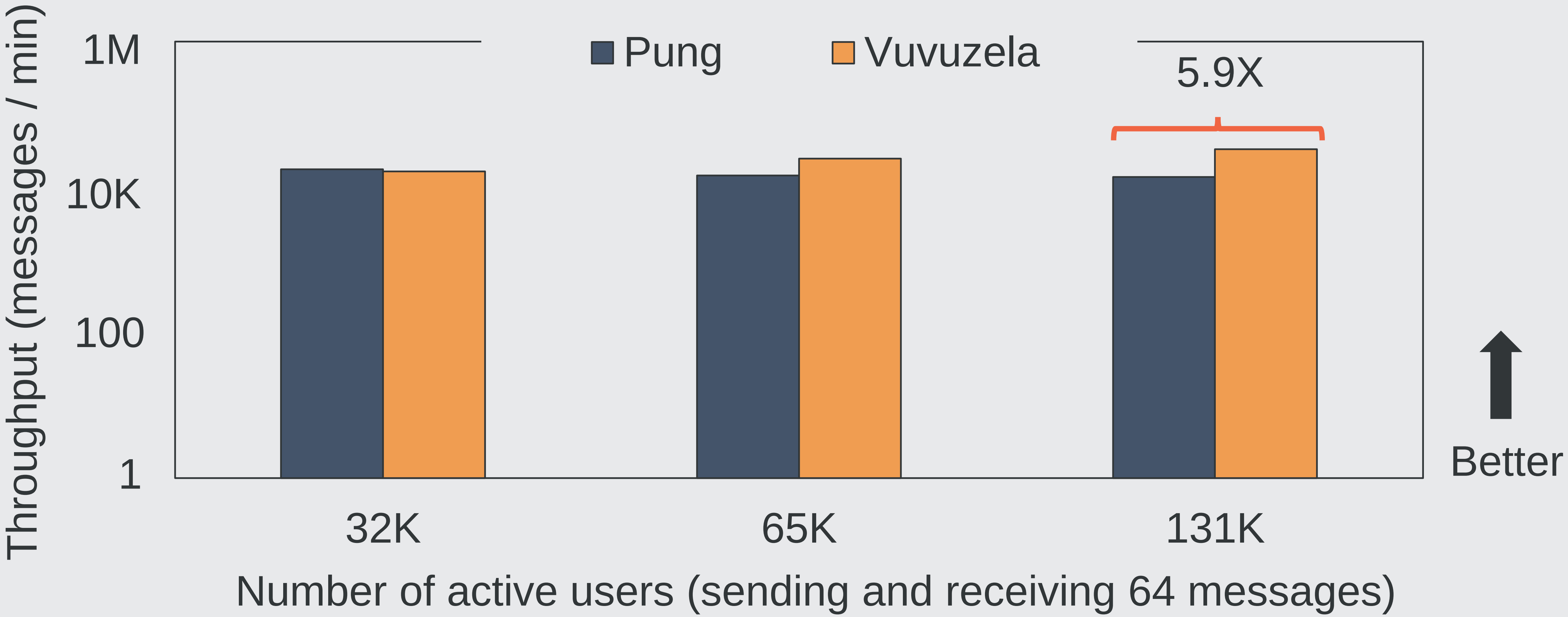
Pung withstands a stronger adversary than Vuvuzela

What is the throughput of Pung when batching?

Pung’s throughput is 6X lower than Vuvuzela



Pung’s throughput is 6X lower than Vuvuzela



Limitations

- High network costs for large batches
- Requires users to know a shared secret (topic of the next talk!)
- No known efficient dialing protocol (also in the next talk!)
- Denial of service is still a problem

In summary, Pung…

- Allows users to communicate privately even if all infrastructure is compromised
- Supports tens of thousands of users
- Introduces a batch procedure that improves efficiency

Code will be available at: https://github.com/sga001/pung

Pung = ROT13(“Chat”)