G(IP)^{2}C

Temporally Isolated IPC with Server-To-Server Invocations

Cédric Courtaud, *Björn Brandenburg*

23/05/12, RTAS'23 San Antonio, TX

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

erc
European Research Council
Established by the European Commission

An Alternative to Locking

Suitable for Mixed-Criticality Systems

Locking protocol

Acquire lock

Prepare food

Release lock

Locking **requires** trust!

An untrusted client can

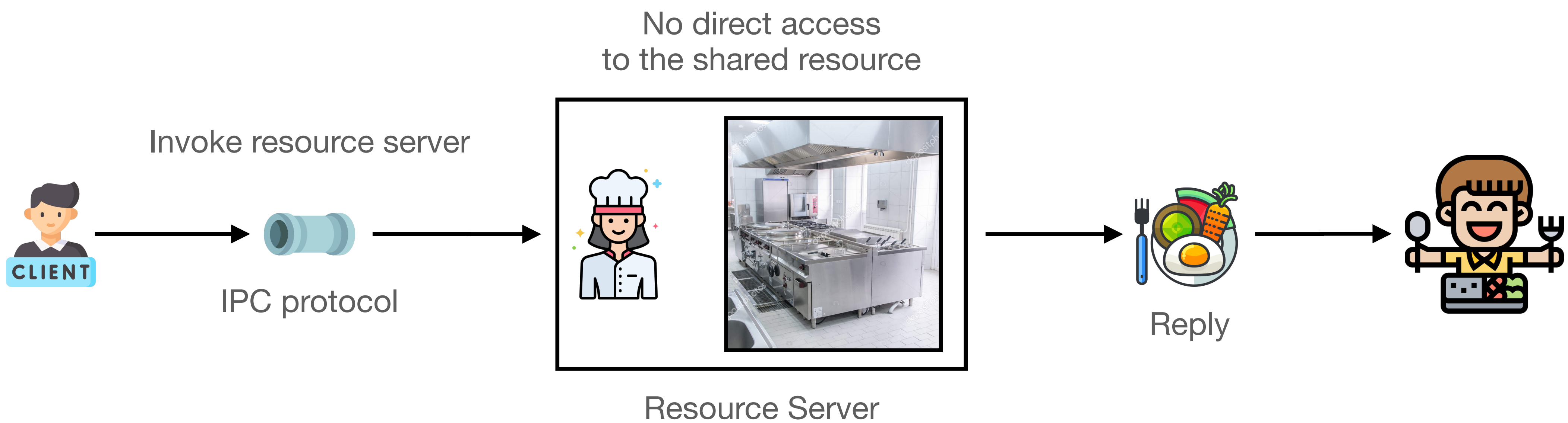
ignore
the lock

Leave the resource
in an inconsistent state

“Forget” to
release the lock

Ruin everybody’s meal

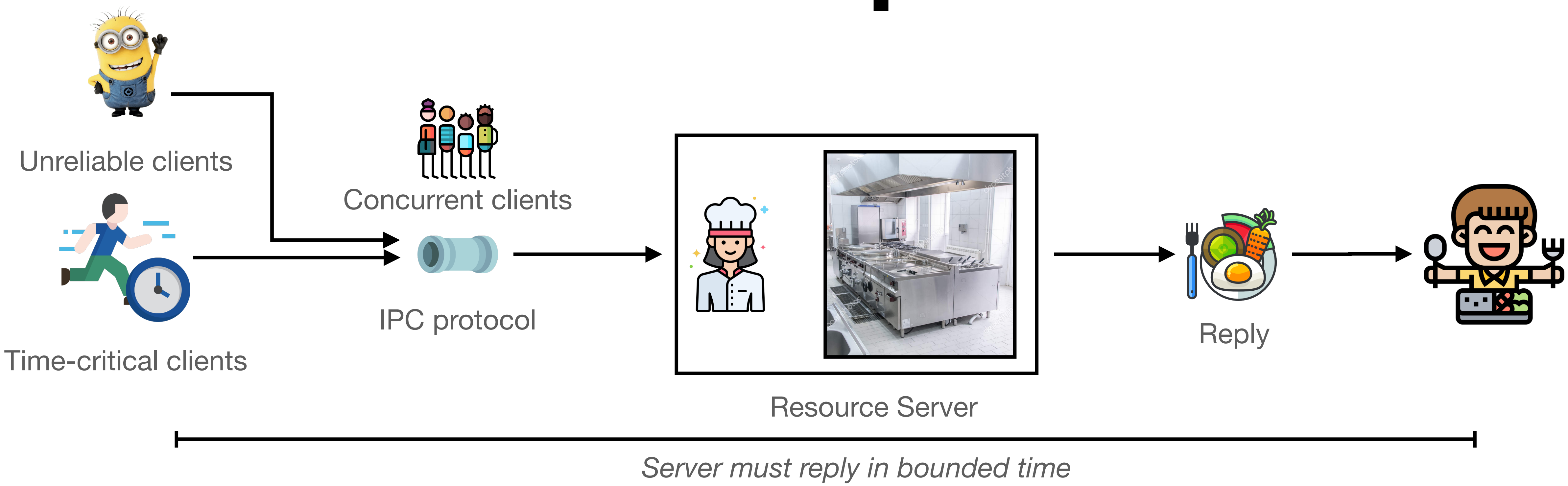
A Better Approach: Resource Servers



Delegating critical section provides **logical** fault isolation.

What about **temporal** isolation?

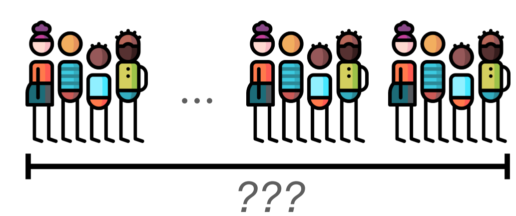
What About Temporal Isolation?



Can we satisfy time-critical clients even if

Some clients try
to **monopolize the server?**

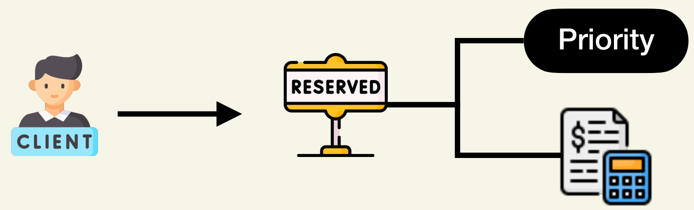
and



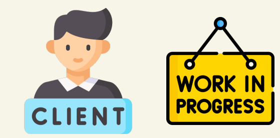
The total number of clients
is **not known** a priori?

Preventing Resource Server Flooding

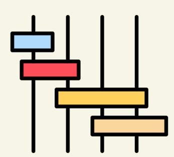
With Reservation-Based Scheduling



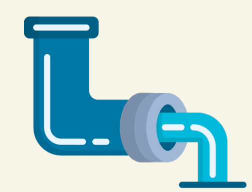
Each client is encapsulated in a **reservation** with a **CPU time budget** and a **priority**



A reservation is **active** if one of its clients has a **pending job.**



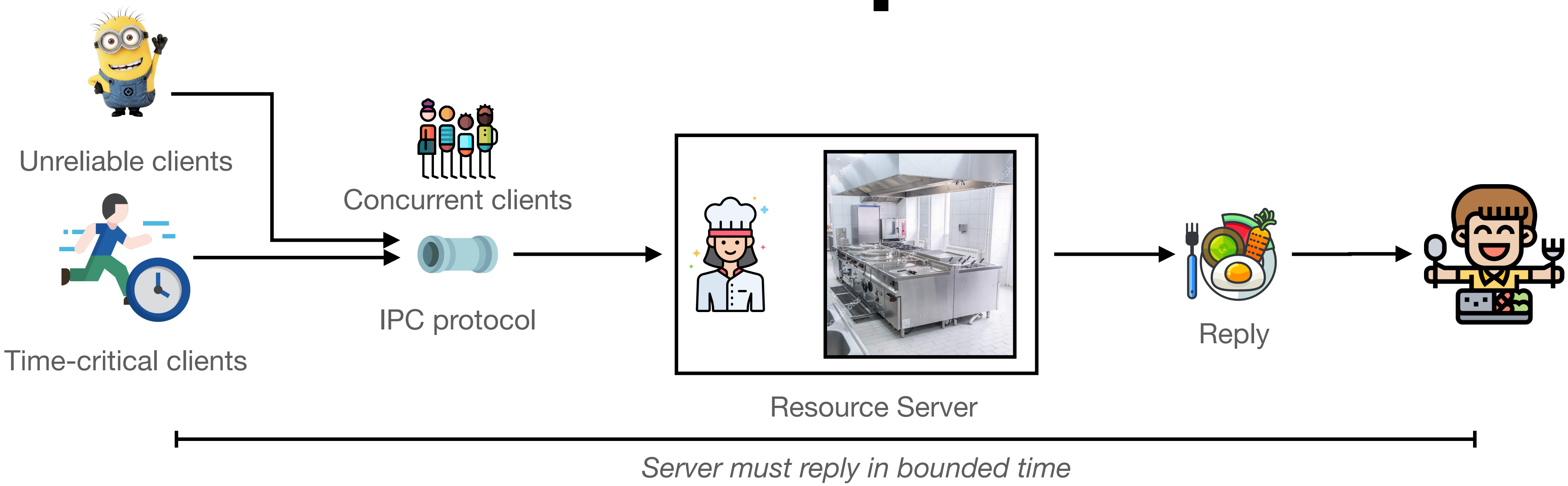
A reservation can be scheduled if it is **active** and has **non-zero budget**.



A **scheduled** reservation **drains budget** at unit rate and **ceases to be scheduled** when its budget is **depleted.**

Clients **cannot** monopolize the processor

What About Temporal Isolation?



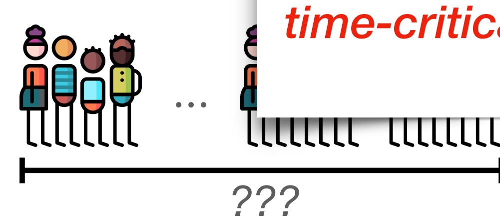
Can we satisfy time-critical clients even if

Yes
(with reservation-based scheduling)

Some clients try to **monopolize the server?**

and

How to provision the budget of time-critical clients?

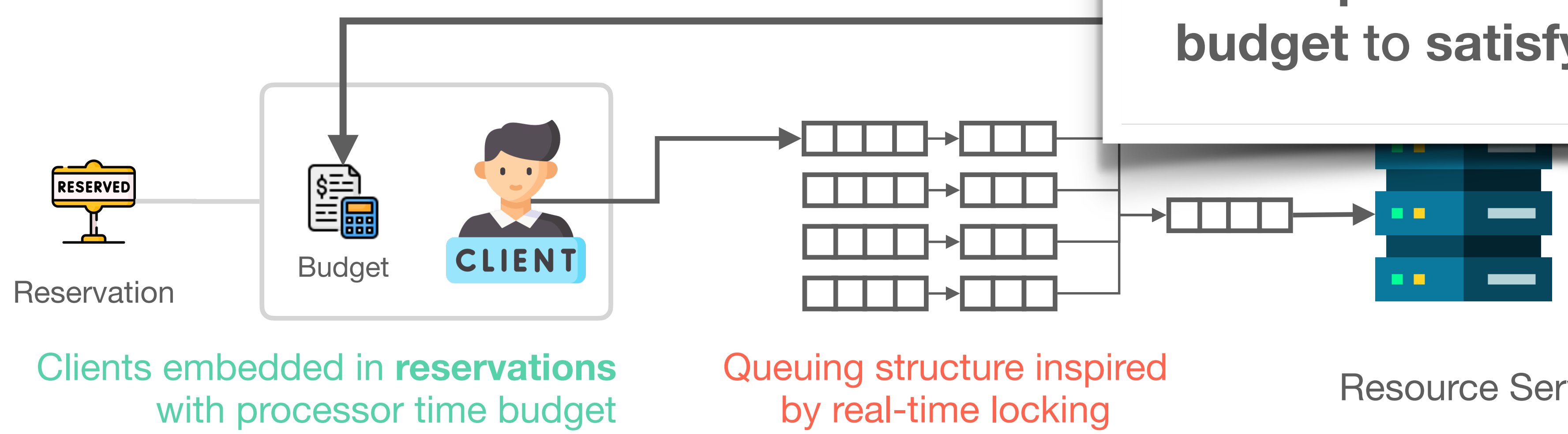


The total number of clients is **not known** a priori?

Prior Work: MC-IPC protocol

"A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems." *RTSS 2014*

Resource servers **migrate** in the reservat



Time-critical clients in reservations can be **provisioned with enough budget to satisfy** their requests

Clients embedded in **reservations** with processor time budget

Queuing structure inspired by real-time locking

Combine techniques from

Reservation-Based Scheduling

Real-Time Locking

Mixed-Criticality Microkernels

😎 Isolation Properties

Bounded Interference

Bound on budget drained for IPC calls That is **independent** of the number of clients

Independence Preservation

No interference from **unrelated** requests

😖 Main limitation

No support for **server-to-server (S2S)** requests

Nesting

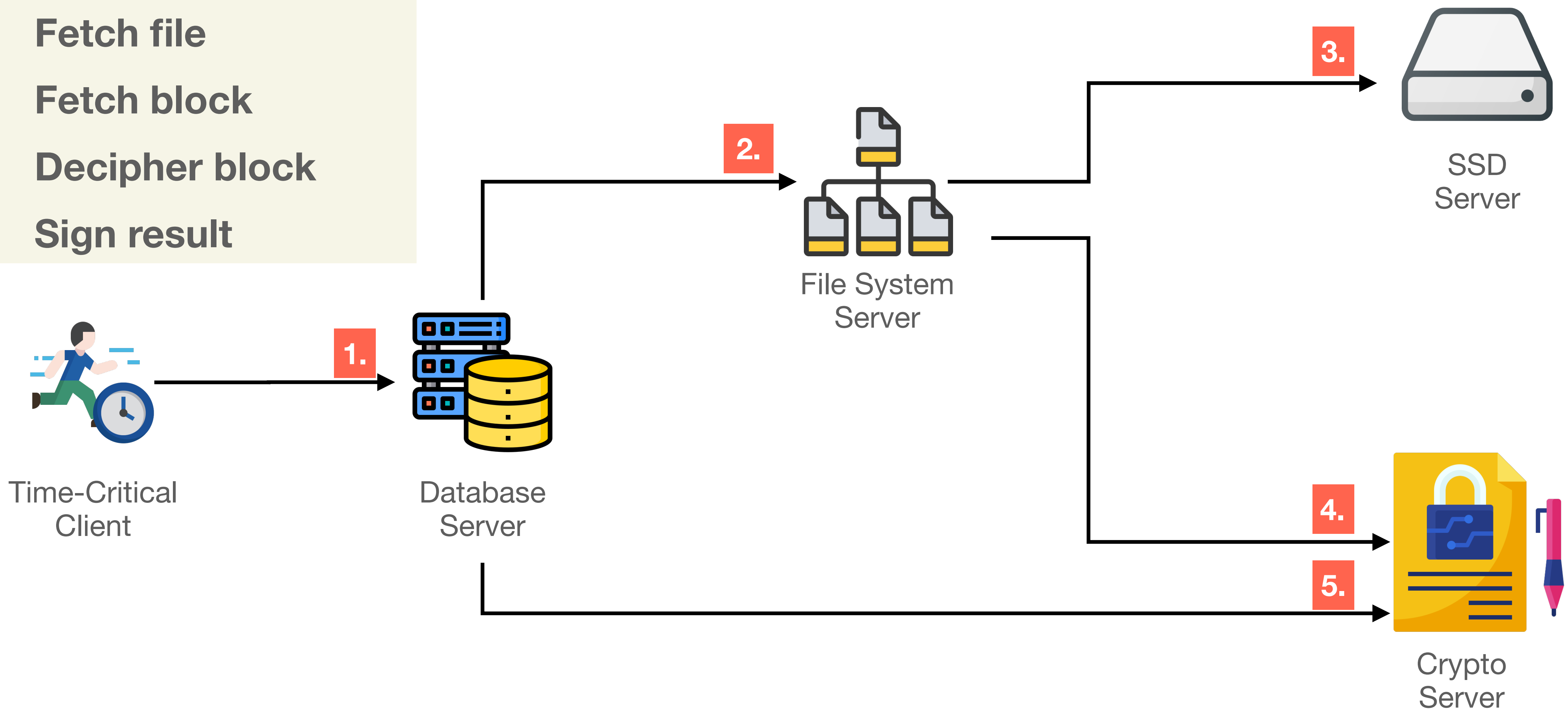
is

EVERYWHERE

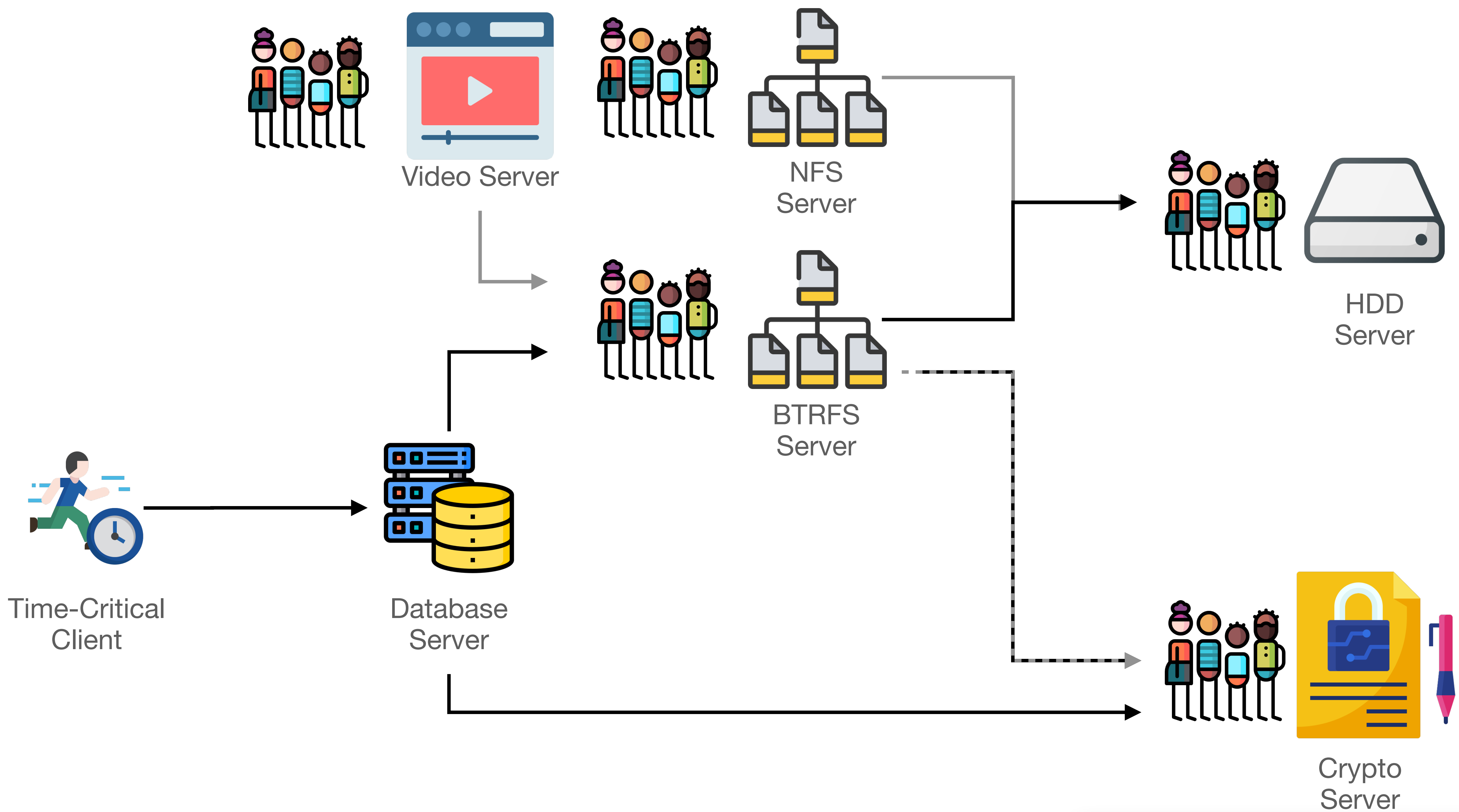
A Motivational Example

Real-Time Secure Database

1. Query database
2. Fetch file
3. Fetch block
4. Decipher block
5. Sign result



Nesting Is Painful



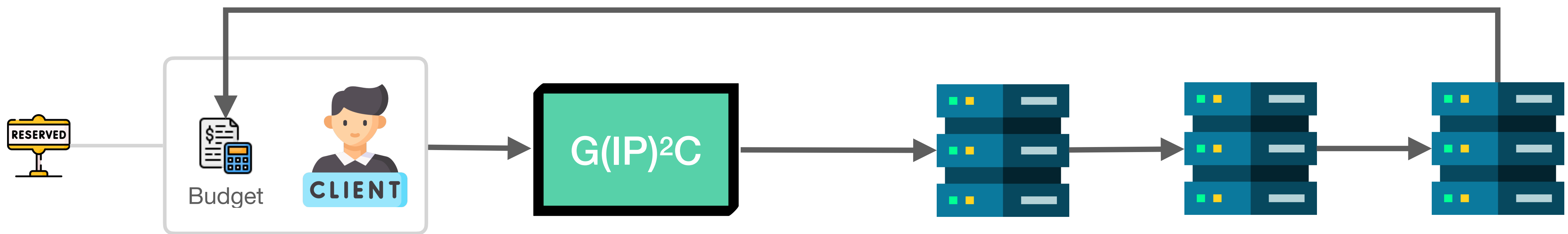
Servers can be called **individually**

Nested requests can be **delayed**

Can lead to **exponential** budget **drain**

Our Contribution

The Group-Independence-Preserving IPC Protocol



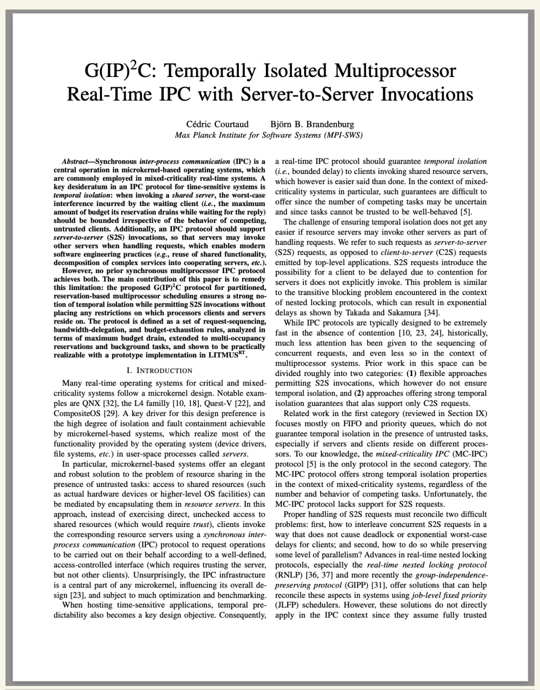
Features

- Supports **S2S requests**
- **Bounded Interference**
- **Group-Independence Preservation**
- **Deadlock-free**

Based on techniques from

Reservation-based scheduling	Nested Real-time Locking		Mixed-Criticality Microkernels
Budgeting	RNLP	GIPP	Passive Servers

Paper Content


G(IP)²C: Temporally Isolated Multiprocessor Real-Time IPC with Server-to-Server Invocations

Cédric Courtaud Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—Synchronous *inter-process communication* (IPC) is a central operation in microkernel-based operating systems, which are commonly employed in mixed-criticality real-time systems. A key desideratum in an IPC protocol for time-sensitive systems is *temporal isolation*: when invoking a *shared server*, the worst-case interference incurred by the waiting client (*i.e.*, the maximum amount of budget its reservation drains while waiting for the reply) should be bounded irrespective of the behavior of competing, untrusted clients. Additionally, an IPC protocol should support *server-to-server* (S2S) invocations, so that servers may invoke other servers when handling requests, which enables modern software engineering practices (*e.g.*, reuse of shared functionality, decomposition of complex services into cooperating servers, *etc.*).**

However, no prior synchronous multiprocessor IPC protocol achieves both. The main contribution of this paper is to remedy this limitation: the proposed G(IP)²C protocol for partitioned, reservation-based multiprocessor scheduling ensures a strong notion of temporal isolation while permitting S2S invocations without placing any restrictions on which processors clients and servers reside on. The protocol is defined as a set of request-sequencing, bandwidth-delegation, and budget-exhaustion rules, analyzed in terms of maximum budget drain, extended to multi-occupancy reservations and background tasks, and shown to be practically realizable with a prototype implementation in LITMUS^RT.

I. Introduction

Many real-time operating systems for critical and mixed-criticality systems follow a microkernel design. Notable examples are QNX [32], the L4 familly [10, 18], Quest-V [22], and CompositeOS [29]. A key driver for this design preference is the high degree of isolation and fault containment achievable by microkernel-based systems, which realize most of the functionality provided by the operating system (device drivers, file systems, *etc.*) in user-space processes called *servers*.

In particular, microkernel-based systems offer an elegant and robust solution to the problem of resource sharing in the presence of untrusted tasks: access to shared resources (such as actual hardware devices or higher-level OS facilities) can be mediated by encapsulating them in *resource servers*. In this approach, instead of exercising direct, unchecked access to shared resources (which would require *trust*), clients invoke the corresponding resource servers using a *synchronous inter-process communication* (IPC) protocol to request operations to be carried out on their behalf according to a well-defined, access-controlled interface (which requires trusting the server, but not other clients). Unsurprisingly, the IPC infrastructure is a central part of any microkernel, influencing its overall design [23], and subject to much optimization and benchmarking.

When hosting time-sensitive applications, temporal predictability also becomes a key design objective. Consequently, a real-time IPC protocol should guarantee *temporal isolation* (*i.e.*, bounded delay) to clients invoking shared resource servers, which however is easier said than done. In the context of mixed-criticality systems in particular, such guarantees are difficult to offer since the number of competing tasks may be uncertain and since tasks cannot be trusted to be well-behaved [5].

The challenge of ensuring temporal isolation does not get any easier if resource servers may invoke other servers as part of handling requests. We refer to such requests as *server-to-server* (S2S) requests, as opposed to *client-to-server* (C2S) requests emitted by top-level applications. S2S requests introduce the possibility for a client to be delayed due to contention for servers it does not explicitly invoke. This problem is similar to the transitive blocking problem encountered in the context of nested locking protocols, which can result in exponential delays as shown by Takada and Sakamura [34].

While IPC protocols are typically designed to be extremely fast in the absence of contention [10, 23, 24], historically, much less attention has been given to the sequencing of concurrent requests, and even less so in the context of multiprocessor systems. Prior work in this space can be divided roughly into two categories: **(1)** flexible approaches permitting S2S invocations, which however do not ensure temporal isolation, and **(2)** approaches offering strong temporal isolation guarantees that alas support only C2S requests.

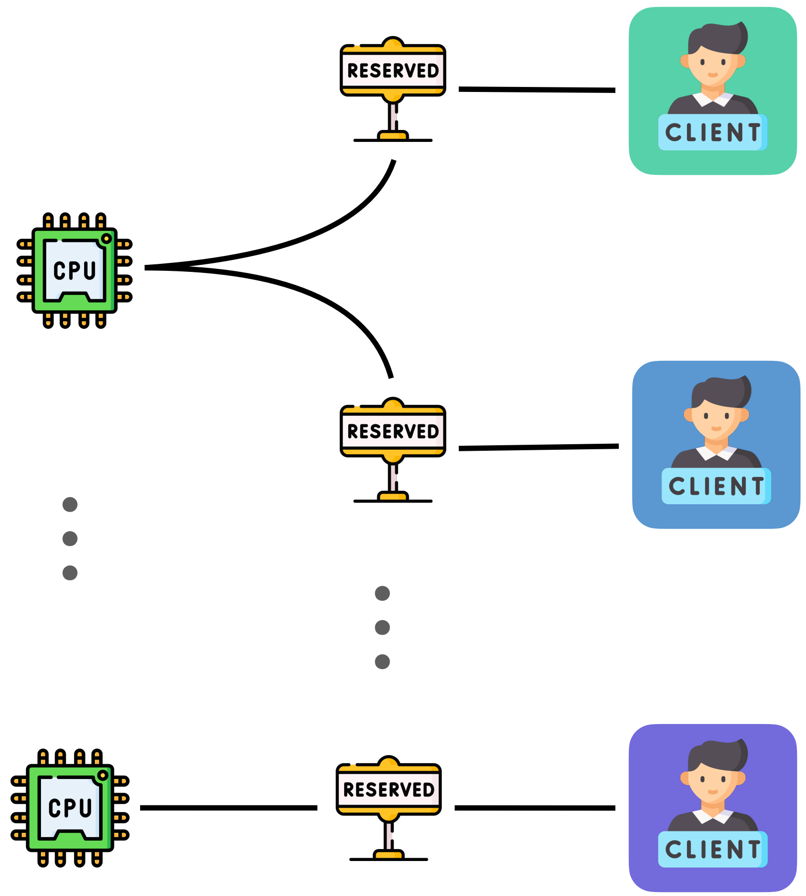
Related work in the first category (reviewed in Section IX) focuses mostly on FIFO and priority queues, which do not guarantee temporal isolation in the presence of untrusted tasks, especially if servers and clients reside on different processors. To our knowledge, the *mixed-criticality IPC* (MC-IPC) protocol [5] is the only protocol in the second category. The MC-IPC protocol offers strong temporal isolation properties in the context of mixed-criticality systems, regardless of the number and behavior of competing tasks. Unfortunately, the MC-IPC protocol lacks support for S2S requests.

Proper handling of S2S requests must reconcile two difficult problems: first, how to interleave concurrent S2S requests in a way that does not cause deadlock or exponential worst-case delays for clients; and second, how to do so while preserving some level of parallelism? Advances in real-time nested locking protocols, especially the *real-time nested locking protocol* (RNLP) [36, 37] and more recently the *group-independence-preserving protocol* (GIPP) [31], offer solutions that can help reconcile these aspects in systems using *job-level fixed priority* (JLFP) schedulers. However, these solutions do not directly apply in the IPC context since they assume fully trusted

Extensions	• Background tasks support • Muti-occupancy reservations
Abortion Rules	• Handling budget exhaustion during IPC
Progress Rules	• Scheduling context transfer
Sequencing Rules	• Concurrent request ordering
System Model	• Main entities involved in the protocol

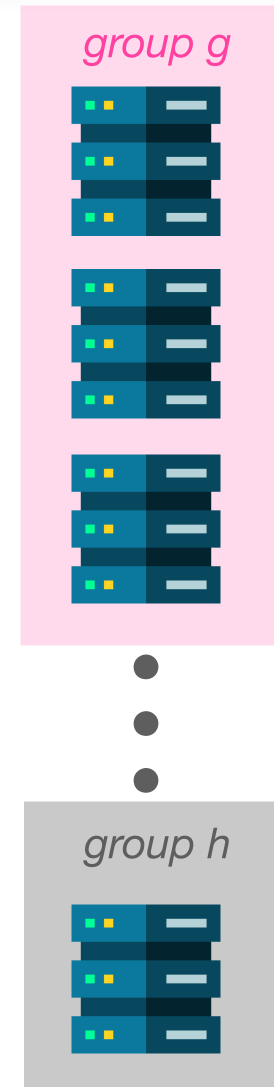
This talk (Sequencing Rules, System Model)

System Model



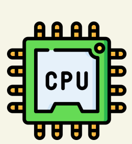
Processors Reservations Clients

Each server belongs to a **server group**



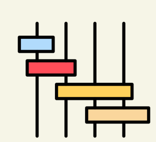
Resource Server

* *Limitation lifted in the paper*



m
processors

n
reservations

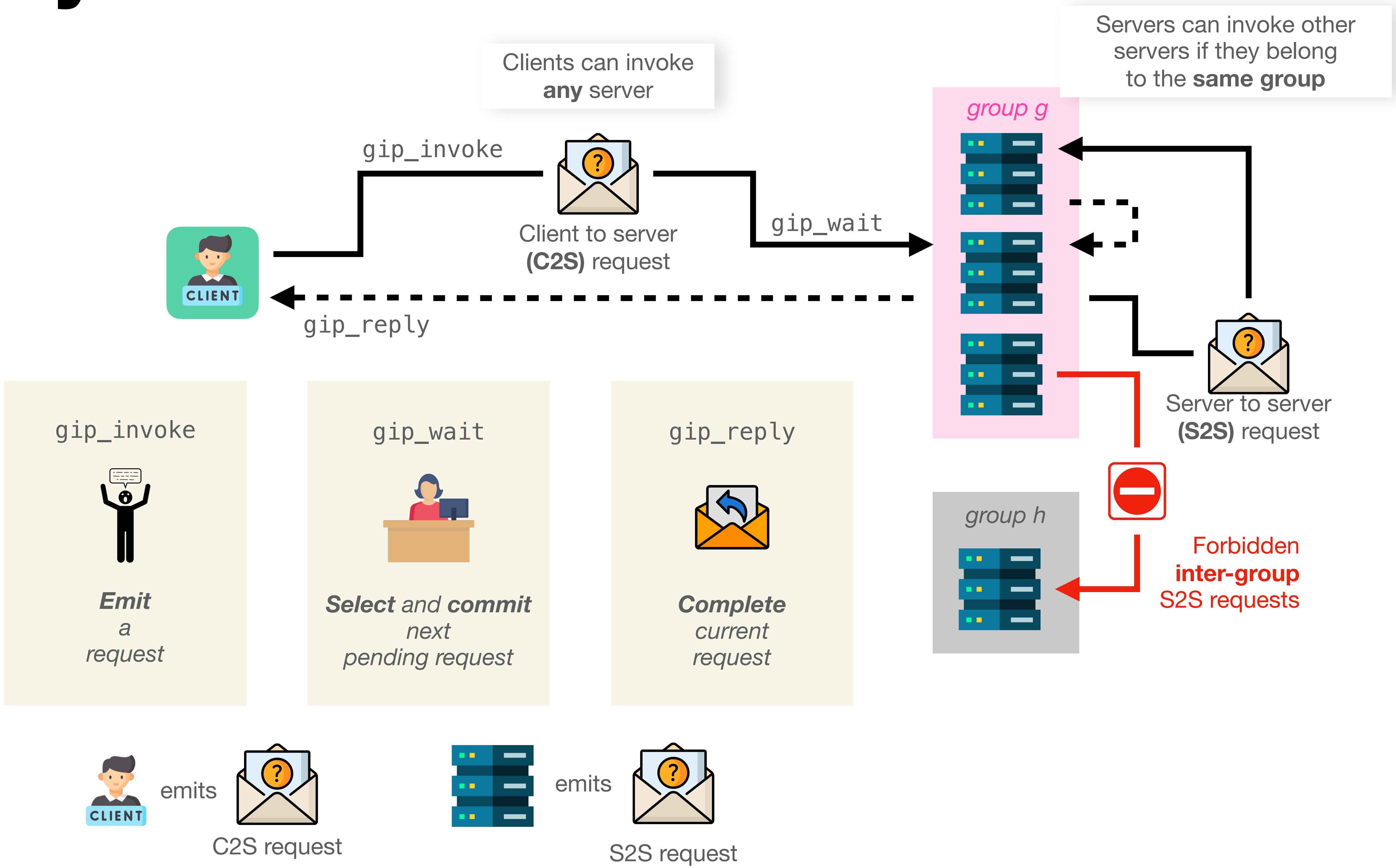


Partitioned
scheduling

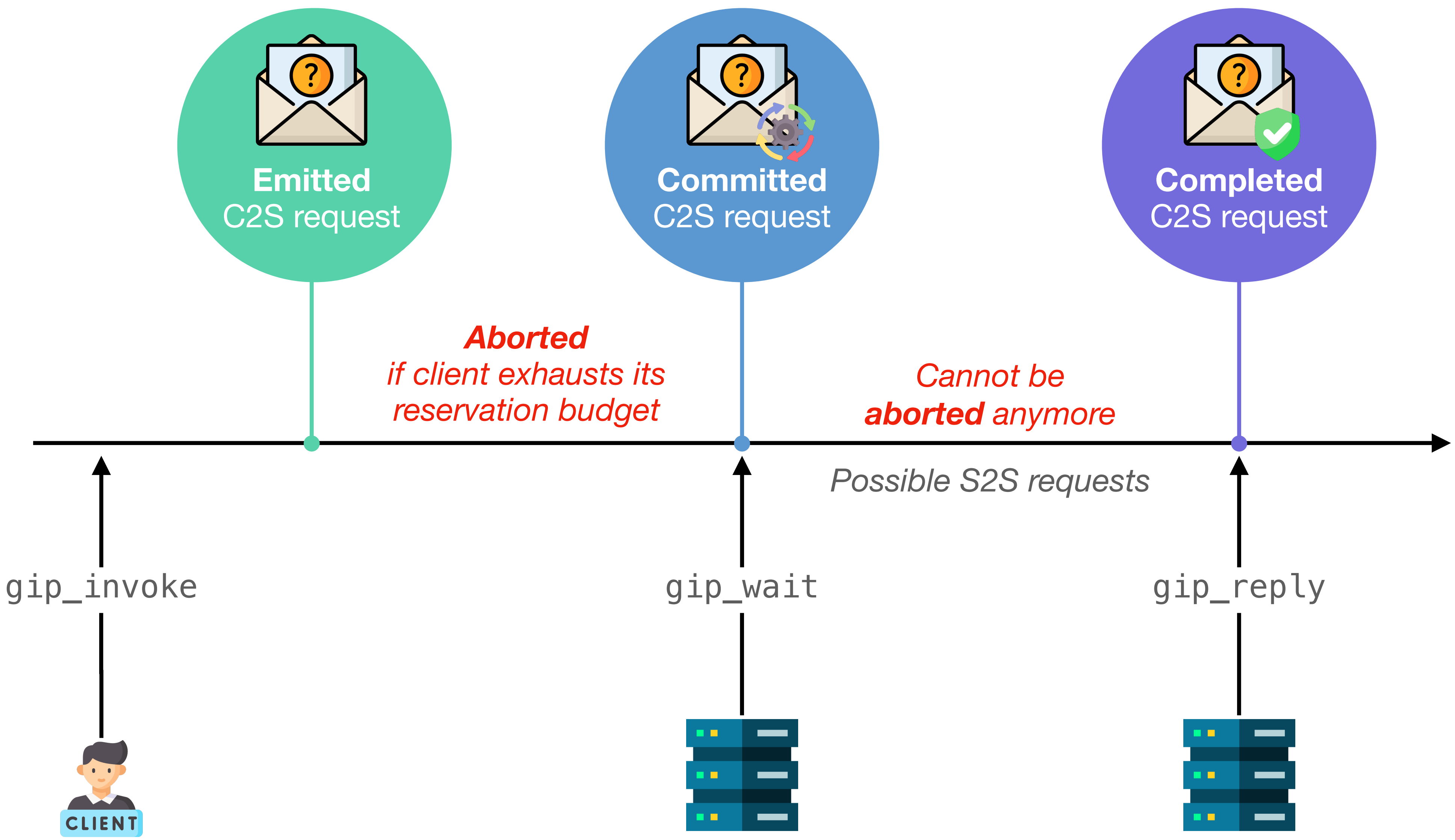
1 ***client***
per reservation*

q
resource servers

Synchronous IPC API

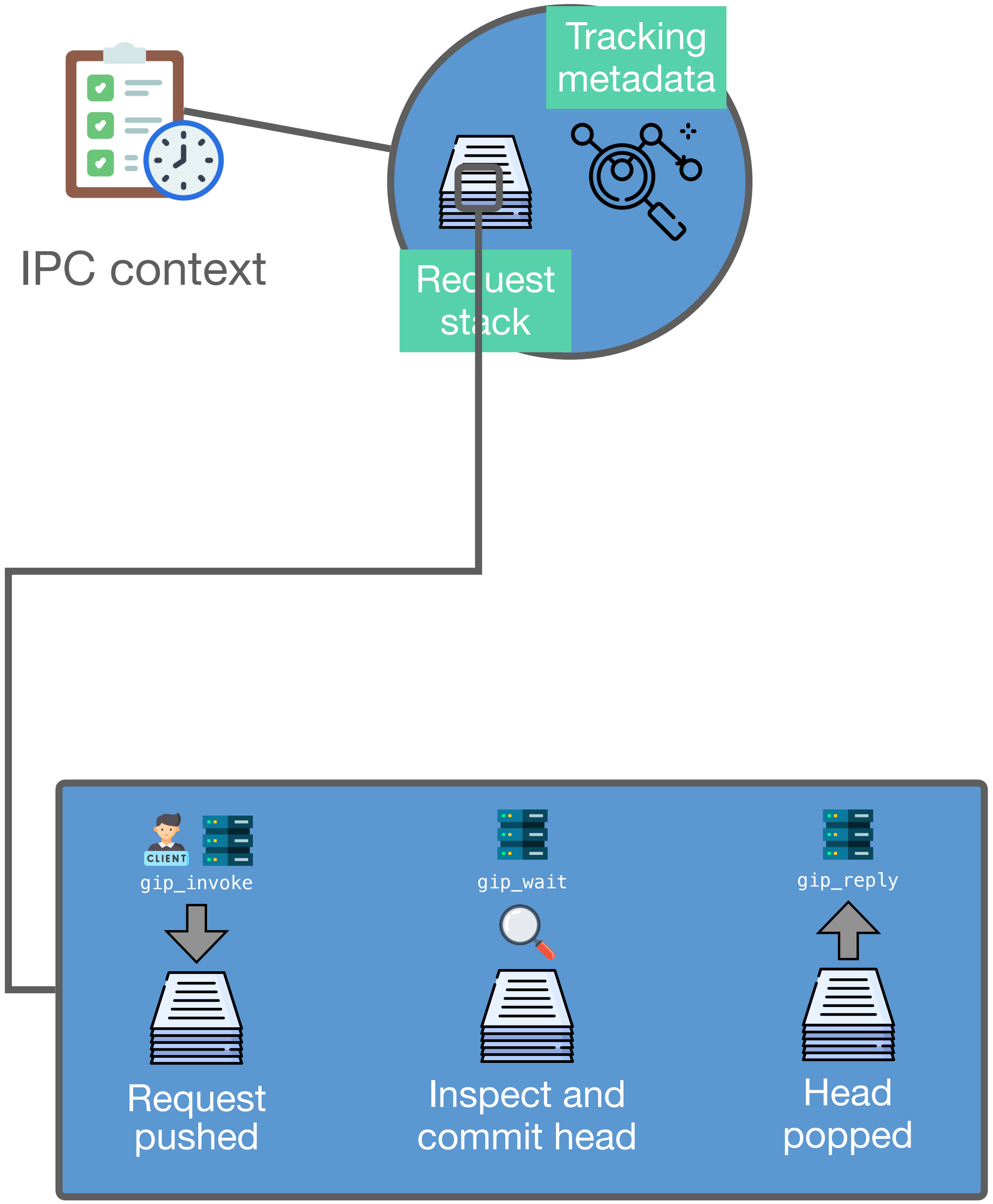


C2S Request Lifecycle



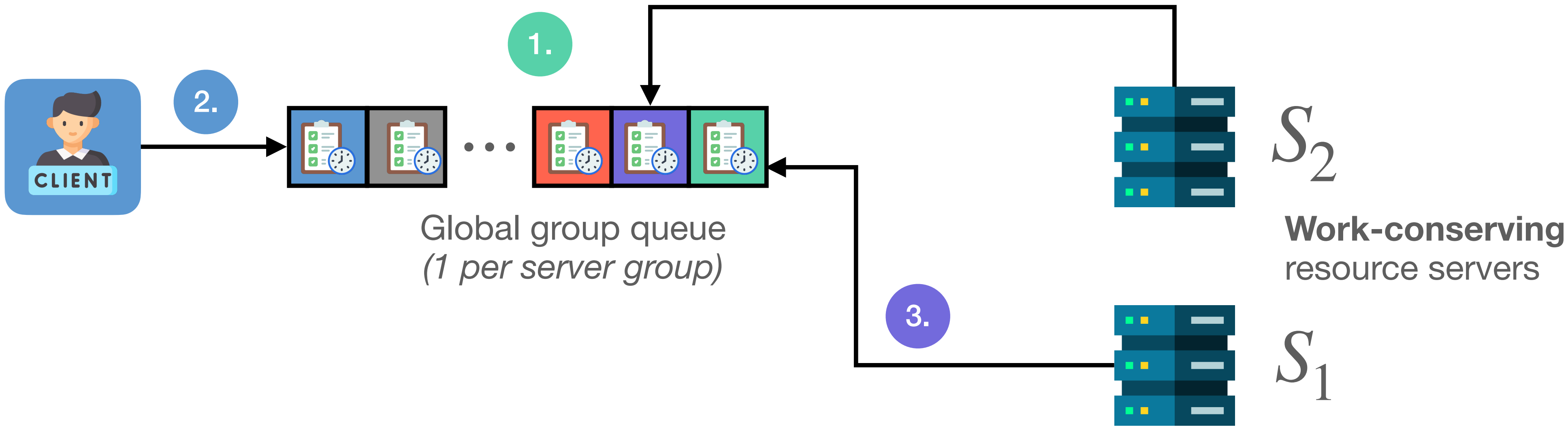
C2S Request Representation

- C2S requests progress tracked by **IPC contexts**
- IPC context contains a **request stack** and **tracking metadata**
- Servers pick requests **from the request stack**



C2S Requests Sequencing

A Straw-Man Approach



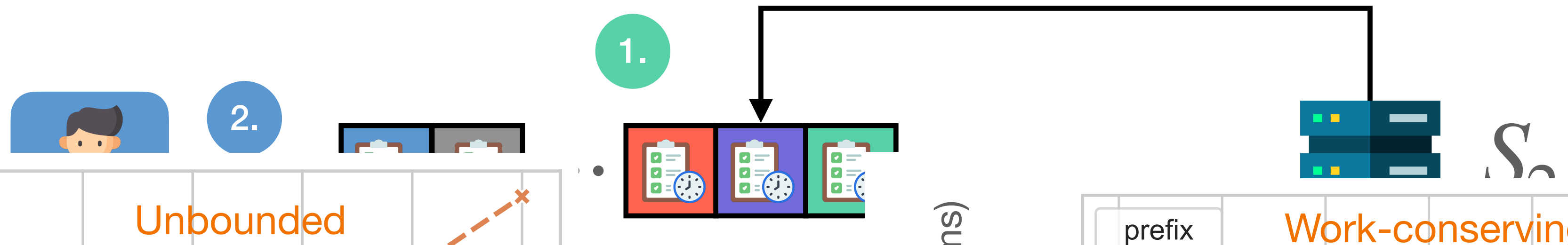
1. Each server group has a **global group queue** of IPC contexts

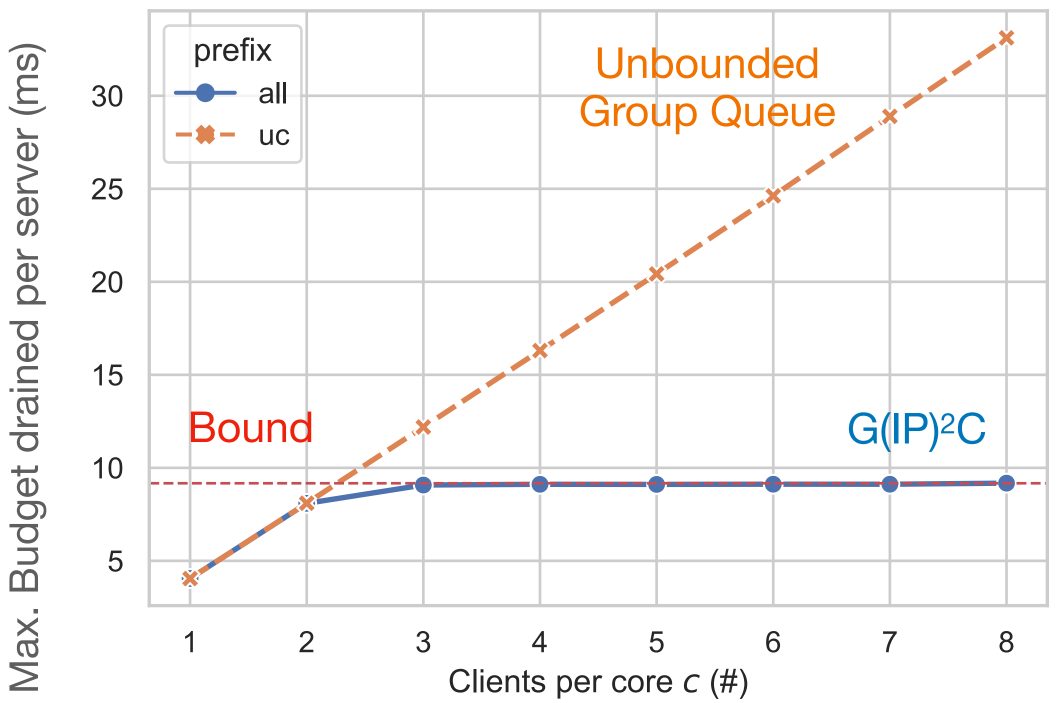
2. IPC contexts enqueued in FIFO order in the group queue.

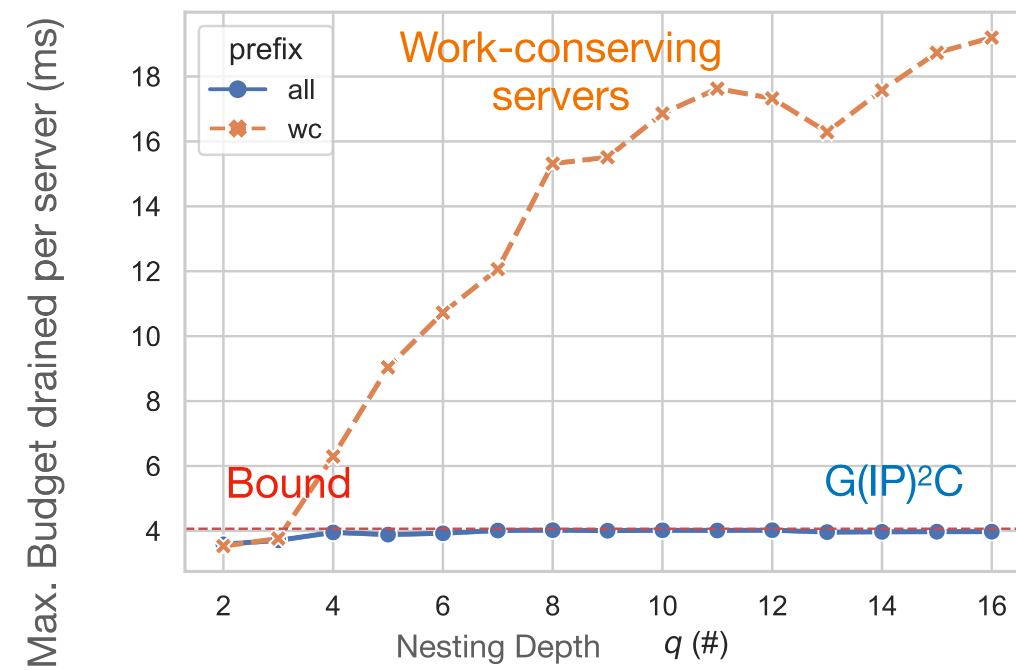
3. Resource servers traverse the group queue and commit requests **as soon as possible**

The Problem with the Straw-Man Approach

How to **bound** the group queue? How to avoid interference from **later requests?**







a **globa**

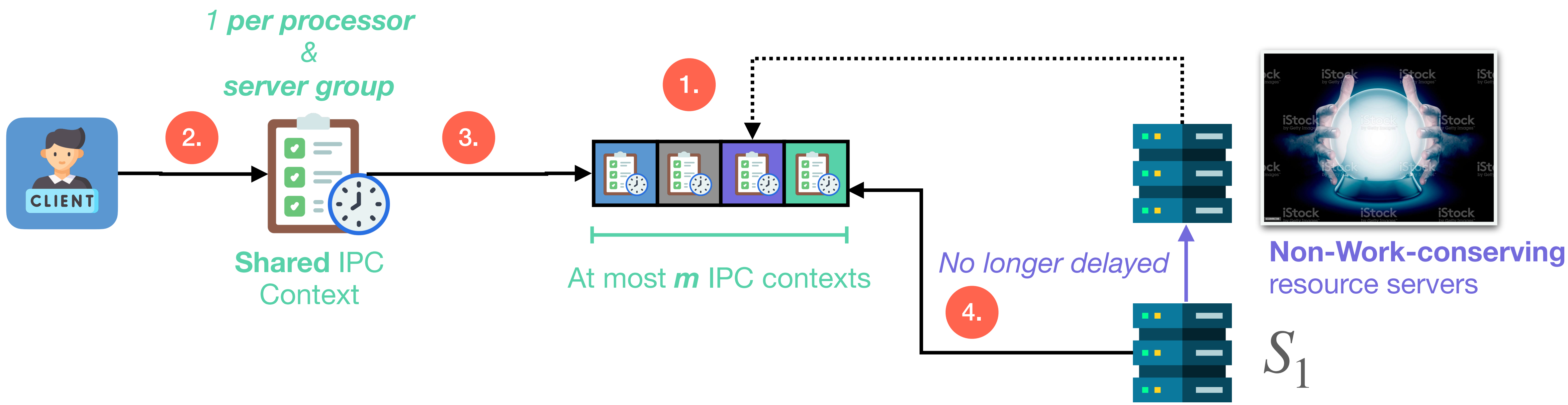
2. IPC contexts enqueued in FIFO order in the group queue.

3. Resource servers traverse the group queue and commit requests **as soon as possible**

Revised Architecture

An RNLP Based Approach

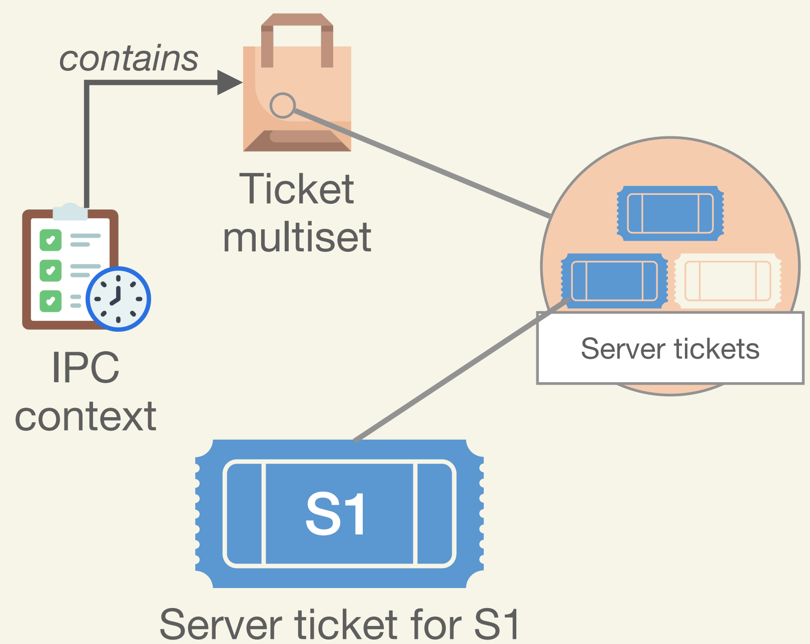
How to **bound** the group queue? How to avoid interference from **later requests?**



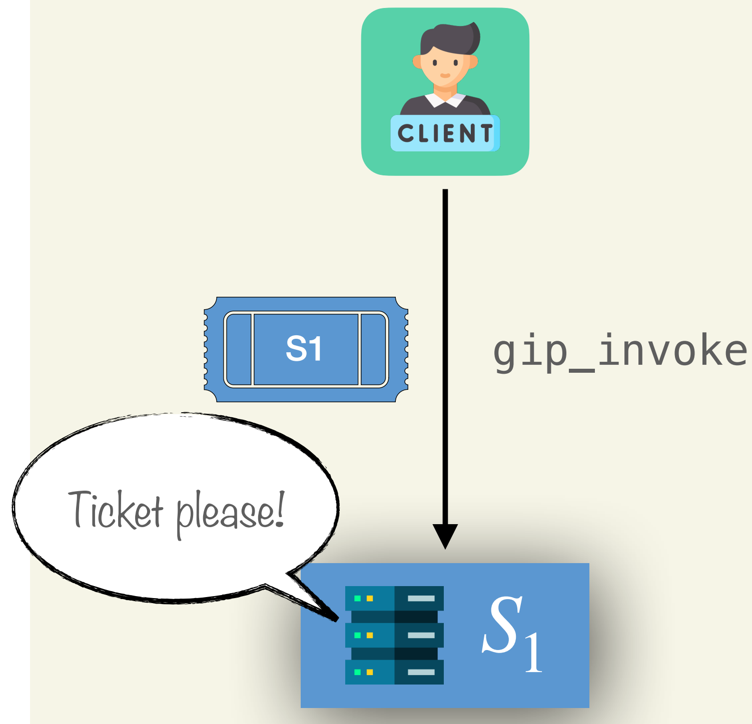
1. Each server group has a **global group queue** of IPC contexts
2. Client **acquires** an IPC context.
3. Once acquired, the IPC context is enqueued in FIFO order in the group queue
4. Resource servers traverse the queue and commit requests **that cannot interfere with earlier ones.** Need to predict the future!

Server Tickets

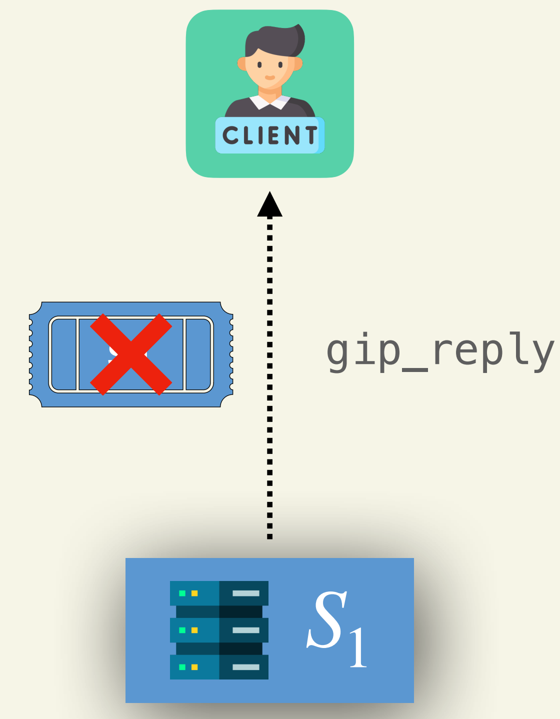
Present and *Future* Resource Server Invocation Tracking



Server tickets are
server-specific tokens
stored in IPC contexts



A ticket is needed
to invoke
a resource server



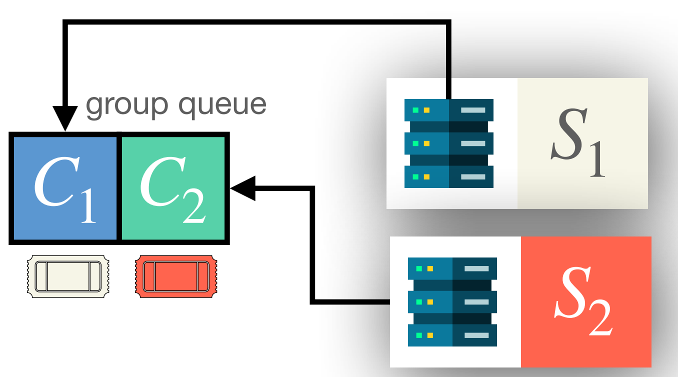
The handled ticket
is consumed
when the server replies

An IPC context with a ticket for S_1 in its ticket multiset
has a request for S_1 or **will have one in the future**

Non-Work-Conserving Resource Servers

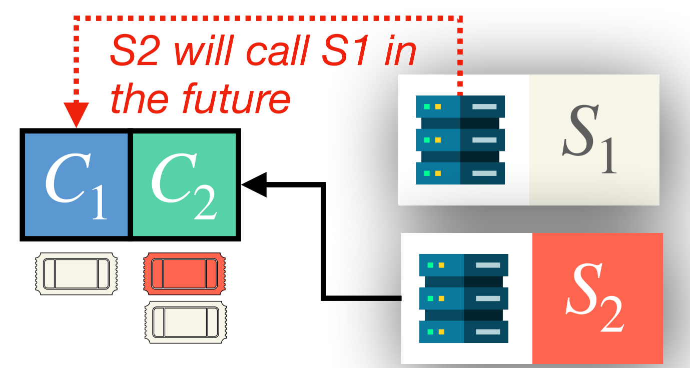
A request is committed only if there is
no preceding IPC context in the group queue
with **overlapping *ticket multiset.***

$tickets(C_1) \cap tickets(C_2) = \emptyset$



S1 and S2 can safely run in parallel

$tickets(C_1) \cap tickets(C_2) = \{\ \}$

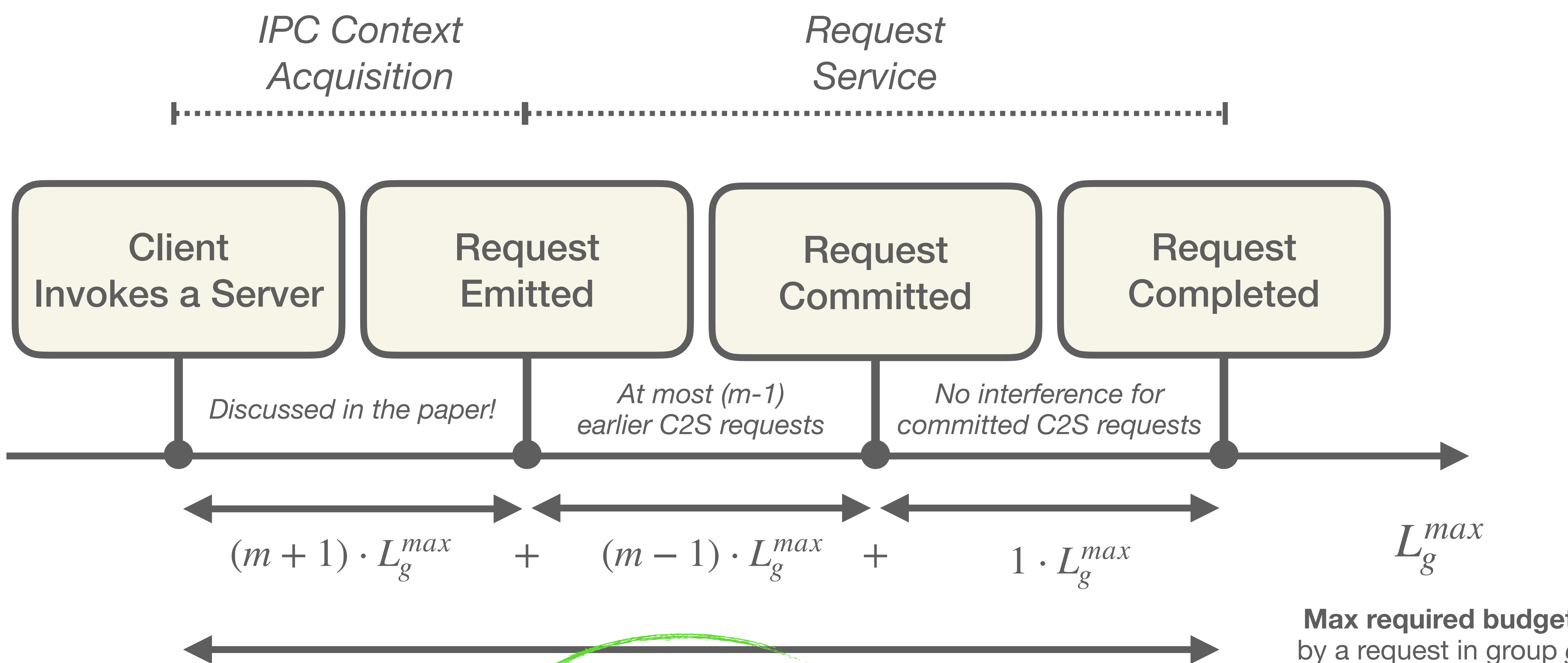


C1 blocks C2's request

Key property

Once committed, a C2S request **is never delayed.**

What budget is needed to satisfy a request?



$$B = (2m+1) \cdot L_g^{max}$$

😃 Does **not** depend on the number of clients! $\Rightarrow$ Allows for **compositional** budget provisioning 🥳

What budget
is drained

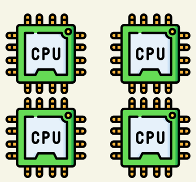
in practice
?

Experimental Setup

LITMUS^{RT}

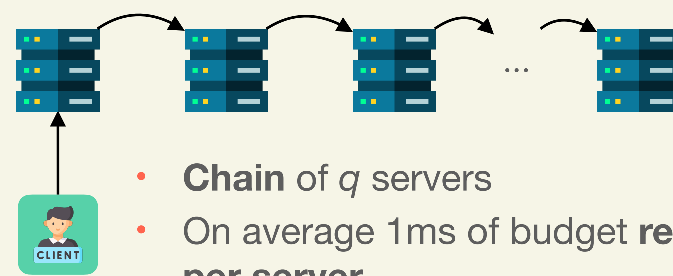
Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Prototype implemented
in **LITMUS^{RT}**



4-core i5-4590
Intel evaluation target

Evaluation Benchmark



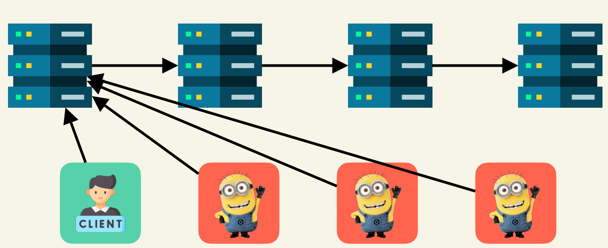
- **Chain** of q servers
- On average 1ms of budget **required per server**

$L_{server}^{max} \approx 1ms$ $\quad q \in [1,8]$

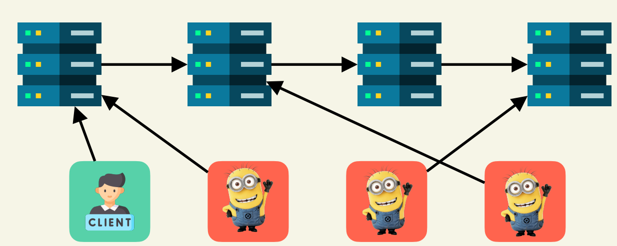
1 **measured** client

(N -1) **stressing** client

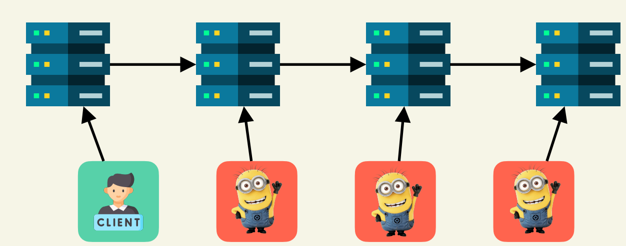
Invocation patterns



Sequential
Adversaries invoke
the **first server**

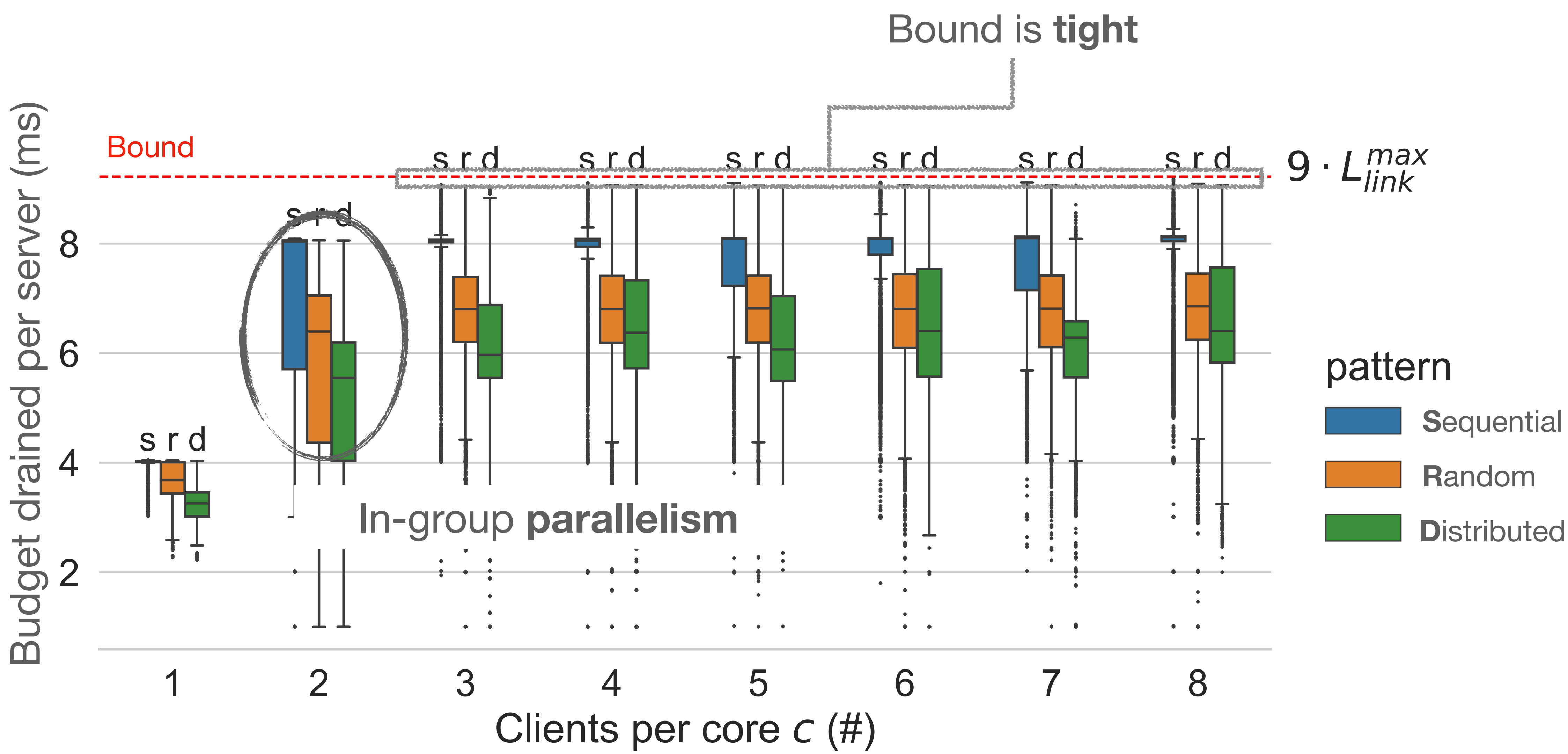


Random
Servers **randomly** picked
by adversaries



Distributed
Servers **evenly** assigned
to adversaries

Our Theoretical Bound is Verified in Practice



🙏Thank you for your attention!

Contribution

G(IP)²C: The first **synchronous IPC protocol** with **temporal isolation** for **S2S invocations.**

Scope

Multiprocessor systems under **partitioned reservation-based** scheduling

Key properties

$(2m+1) \cdot L_g^{max}$ Bounded Interference	Group-Independence Preserving	Deadlock Free

There is more in the paper!

- Full proof
- Progress rules
- Abortion Rules

Limitations

Requires **careful resource partitioning**

Higher Runtime Overhead than simpler protocols

Extensions

Support for **background jobs**

Multi-occupancy reservations