# NEXT GENERATION INTERNET

## Availability

Christian Grothoff

4.4.2025

# Learning objectives

How to architect systems for high-availability?

What are specific considerations for the network setup?

What are specific considerations for HTTP servers?

What are specific considerations for the application logic?

What are specific considerations for databases?

What are specific considerations for monitoring?

How to know your limits?

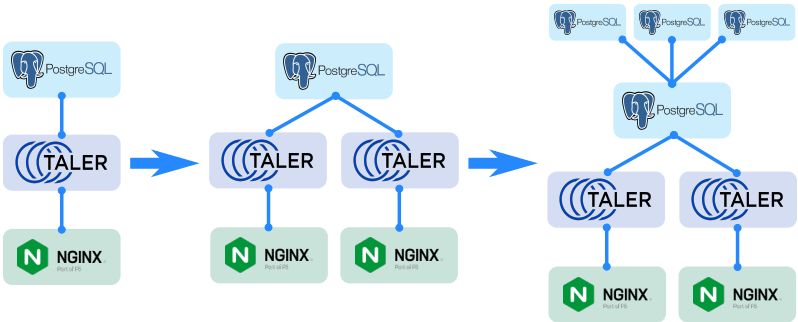
# How to architect systems for high-availability?

# High-availability principles

Use:

- redundancy at all levels:
  - RAID (5, 1+1, 1+1+1), ECC RAM, redundant power supplies, ...
  - redundant servers
  - multiple power sources (grid, generator, battery, UPS)
  - multiple data centers
- a layered architecture

# Horizontal distribution

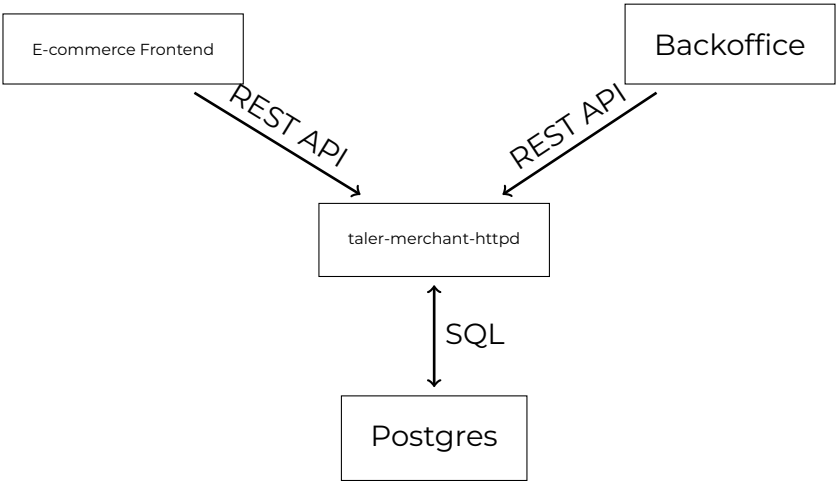


# High-availability principles

- Know your (performance) targets and limits
- Know (monitor) your load (and availability)
- Know your interactions and dependencies

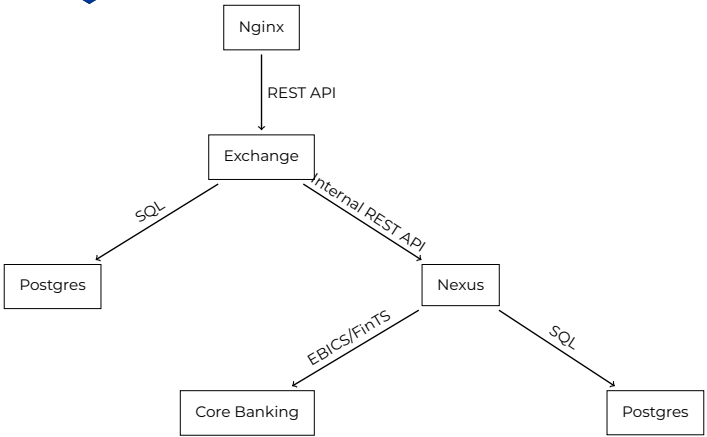
# Taler example

## Merchant architecture



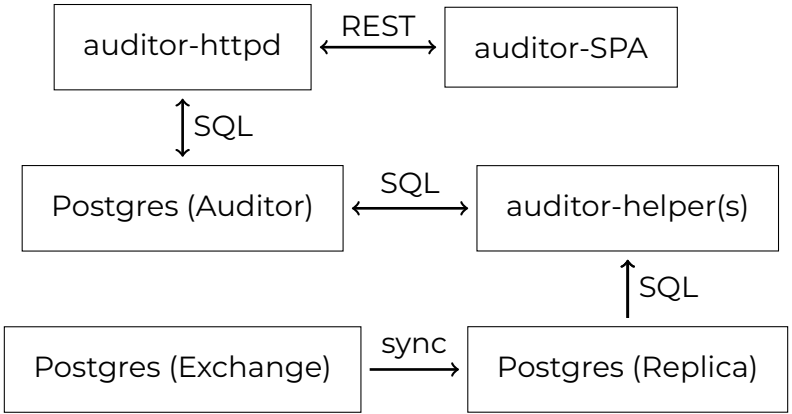
# Taler example

## Bank architecture



# Taler example

Auditor architecture



## Part II: What are specific considerations for the network setup?

# Network setup for availability

- IPv4 + IPv6 dual stack: some users today are only on IPv4, others only on IPv6
- Good data centers have multiple, redundant up-links via different providers
- Data centers should have backups for everything: cooling, battery power, fuel for on-site generators, etc.
- Even better are multiple data centers; data centers do burn…
- High-availability hosters monitor for disasters and migrate operations out of dangerous areas

# DNS is critical

## Microsoft learned the hard way, twice

**Network Connectivity**

Engineers are currently investigating DNS resolution issues affecting network connectivity to Azure services. More information will be provided as it becomes available.

Refresh every

2 minutes

Good Warning Error Information

Americas | Europe | Asia Pacific | Africa | Azure Government

| PRODUCTS AND SERVICES | NON-REGIONAL* | EAST US | EAST US 2 | CENTRAL US | NORTH CENTRAL US | SOUTH CENTRAL US | WEST CENTRAL US | WEST US | WEST US 2 | CANADA EAST | CANADA CENTRAL | BRAZIL SOUTH |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IMPACTED SERVICES | | | | | | | | | | | | |
| Network Infrastructure | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| COMPUTE | | | | | | | | | | | | |
| Azure VMware Solution by CloudSimple | | ✓ | | | | | | ✓ | | | | |
| Virtual Machines | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

# Firewalls?

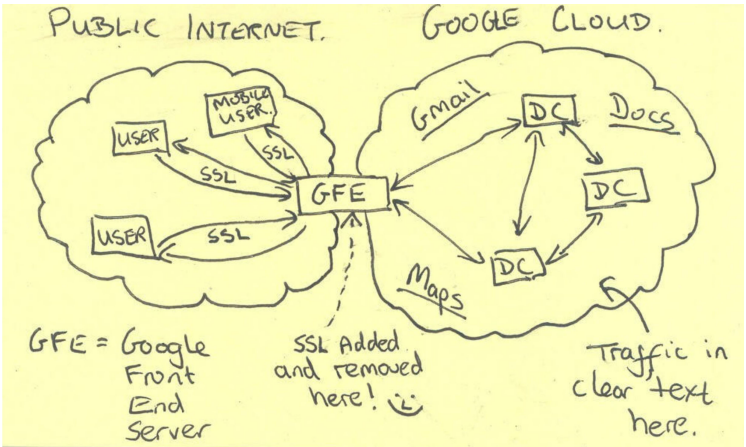
- Firewalls add complexity, misconfiguration, hardware- and software-failures can all harm availability
- Deep packet inspection firewalls parse all types of untrusted input, and they must do so quickly and often with high priviledges; this is one of the most problematic types of software one could ever use
- An attacker who successfully takes over your firewall does not merely defeat this layer of security, but is now in a great position to monitor your network.

$\Rightarrow$ Firewalls are dangerous.

# Good network access control

- First, only use IP if needed. UNIX domain sockets are faster and more secure for inter-process communication on the same host! You can also run HTTP over a UNIX domain socket.
- Second, if software does not support UNIX domain sockets, at least binding to loopback (`::1`) should be widely supported. Use `netstat -npl` to check which addresses local services are bound to.
- If network access is required but only from a particular host, *maybe* a simple host-based firewall can be useful as an additional security layer.
- Alternatively, use TLS and/or a Wireguard VPN to build a secure tunnel. TLS client-certificates could be used to authenticate the client.

# Don’t be one of these guys...



# DNS and DNSSEC

- DNS with high TTL values improves caching and performance, but may increase down-time for unplanned migrations to new IP addresses
- Including multiple A/AAAA records and randomizing their order in DNS responses can be used for load-balancing among front-end servers
- DNSSEC is the only viable defense against DNS cache poisoining attacks. Use it to ensure your users are not sent elsewhere by DNS!

## Part III: What are specific considerations for HTTP servers?

# HTTP service considerations

- Cache-control is your friend, both for performance and to make CDNs effective against DDoS.
- Ensure your applications enable caching with long durations and good ETags whenever possible.
- Defending against a DDoS requires enough bandwidth and computational power to handle requests. You can get bandwidth from a CDN for resources that can be cached. Make sure you have enough computational resources to handle non-cachable requests!
- Scalable system architecture is key: ideally you can add resources only for the duration of the DDoS.
- In addition or as an alternative to DNS-based load balancing, HTTP(S) reverse proxies can also be used for load balancing.

# Benchmarking

- Simplistic benchmarks like `ab` or `h2load` mostly measure your HTTP server, which is rarely the bottleneck.
- Good benchmarking requires generating **realistic** load across your API. So use your actual client applications, or at least derive traffic patterns from real-world load on your system.
- It is also useful to do a **worst-case analysis**, trying to find out what the most expensive requests are, especially in case an attacker tries to take you down based on those.

## Part IV: What are specific considerations for the application logic?

# Scaling application logic

- Vertical scaling (using multiple CPU cores) is the obvious solution for improving application scalability.
- Multi-threading significantly increases application complexity, and also does not work for horizontal scaling. Thus, it should be used rarely (such as for expensive cryptographic operations) if at all!
- Using multiple **processes** (one per core) is marginally more expensive, but much simpler, works for vertical and horizontal scaling, and generally more secure!

# Architect for least priviledge

- Software engineers often debate between micro-services and monoliths. Both are wrong!
- Monoliths are bad because they tightly couple too many business concerns.
- Micro-services are bad because they are too simple and thus require complex orchestration.

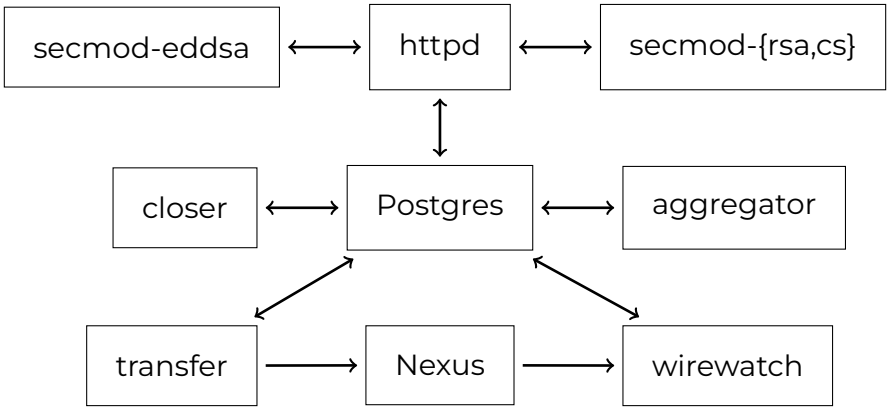
# Architect for least priviledge

- Software engineers often debate between micro-services and monoliths. Both are wrong!
- Monoliths are bad because they tightly couple too many business concerns.
- Micro-services are bad because they are too simple and thus require complex orchestration.

⇒ Instead, architect for **least priviledge**: if a business concern requires specific rights distinct from other business concerns, that is a good reason to move it into a separate component!

# Taler example

## Exchange architecture



# Minor havoc should be mandatory...

- Havoc is a technique where faults are regularly injected into a system to ensure programmers develop fault-tolerant software
- `systemd` can and should be used to limit service process lifetimes (to say `1h`). This prevents minor memory leaks and memory fragmentation issues from becoming relevant in production. Postgres also does **not** like long-lived client connections.
- Ensure to configure systemd to **auto-restart** services (also good after crashes).

# Taler's fork-close-systemd-listen trick

- Systemd can `listen` on a socket and pass the already open listen socket to an application process. This has the advantage that the listen socket remains open (and the kernel answers SYN packets) even if the application process is briefly down.
- When a Taler service is asked (SIGTERM) to terminate, it `closes` its listen socket(s), forks and the child continues to handle active clients and exits once those requests are finished. The original process exits immediately.

## Part V: What are specific considerations for databases?

# Database performance

- Database performance is largely determined by writing good queries that make good use of indices.
- Indices can also be expensive. Learn about **partial indices**.
- When using a separate database host, and especially when sharding, the latency to the database can matter. Here, using **stored procedures** instead of multiple SQL statements can minimize performance issues arising from latency.

# Database versioning

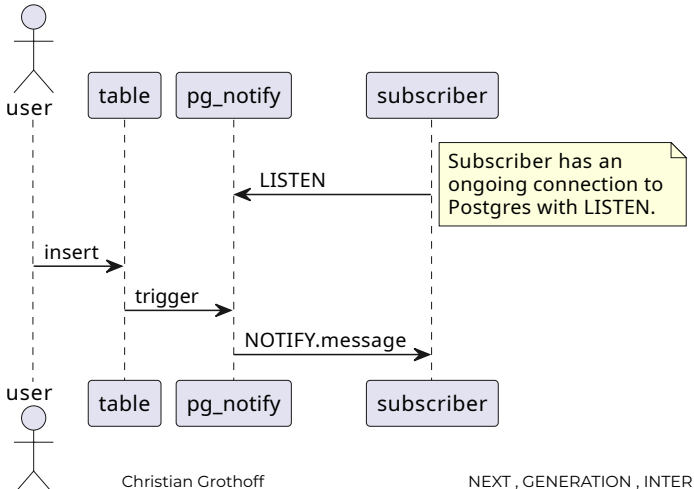
- Database schema **will** evolve over time as application requirements change
- Database versioning should be used to track the current state of the database schema.
- The database version should be stored **within** the database itself and updated with the transaction that does the schema migration.
- `https://www.depesz.com/2010/08/22/versioning/` has code for a good simple approach.

# Least priviledge ...

## ... also applies to databases

- GRANT only the required rights to each DB client
- In particular, schema updates should probably only be done by the DB owner

# Postgres as a pub-sub service



# Postgres as a pub-sub service

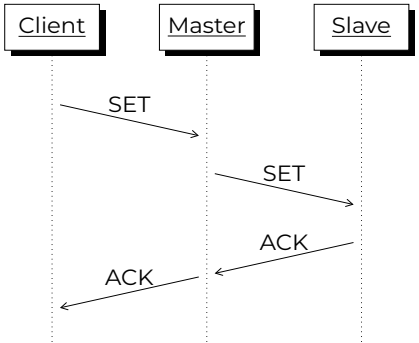
## Implementation concerns

- In some programming languages, LISTEN must be done in a separate database connection.
- NOTIFY and LISTEN are very high performance operations.
- NOTIFY is transactional, so notification is guaranteed.
- LISTEN and NOTIFY are great mechanisms for inter-process communication between multiple processes that use the same database!

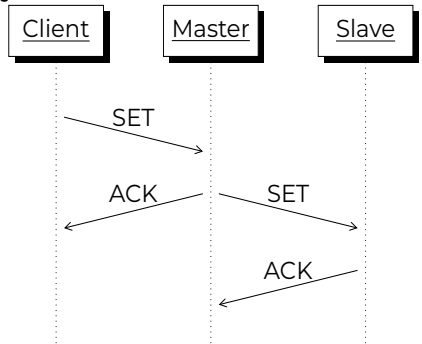
# Database backups

## Synchronous vs. asynchronous replication

Synchronous:




Asynchronous:



## Part VI: What are specific considerations for monitoring?

# Network monitoring

Monitor at least:

- IP connectivity
- DNS resolution
- Service availability

and if possible from **external** infrastructure.

Automatically **notify** staff of incidents.

# Monitoring latency

Application latency is often best monitored at the HTTP reverse proxy:

- reverse proxies widely support logging request latency
- the most interesting information – which endpoint – is available
- network latency to the client is excluded, which is good as it is usually outside of your control
- no need to send telemetry data from the client (less bandwidth, more reliable, simpler architecture, better for privacy)

# Application-specific monitoring

- Design your applications to enable monitoring.
- Log ERRORS if interventions are urgent, WARNINGS if investigations are in order.
- Have counters for key events.
- Export application-specific performance metrics.

# Monitor for slow queries

- Use “slow query” logging of your database to detect problematic queries and then ANALYZE them.
- Query optimizers can be fickle: a query may perform well at first, but then due to changes in database-internal metrics a query optimizer may switch to a much **worse** execution plan.

# System-level monitoring

On each host, you want to monitor:

- System load (CPU load, memory utilization, disk utilization, disk IO load, bandwidth)
- Running processes (total, application specific services, state)
- Core dumps / crashes
- Other metrics (depending on your application)

# Visualize

## A picture is more than 1000 data points

- Plot your data. This is the only way to **quickly** spot problems.
- Create dashboards. You will need to not just plot individual data points, but also spot correlations.
- Combine data sources: host, network, application monitoring; logging, performance metrics

⇒ Use tools like Grafana or `https://nagios.org` or Zabbix

Example dashboard

# Part VII: How to know your limits?

# Microbenchmarks

- Microbenchmarks give you a critical **upper bound** on system performance.
- If your database takes 300ms per query and does at most 20 queries in parallel, what is your maximum transaction rate?
- If a transaction requires 50 kilobytes of bandwidth and you have 1 GB/s, what is your maximum transaction rate?
- `https://bench.cr.yp.to/` provides extensive benchmarks for cryptographic primitives. Study it to know how fast your cryptographic routines will be!

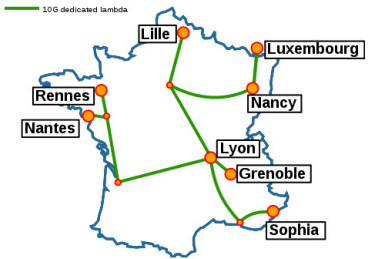
# Burn your systems

Before going into production, burn your systems:

- `memtest86+` (RAM)
- `stress` (CPU, RAM)
- `stress-ng` (CPU, RAM)
- `lookbusy` (CPU, RAM, disk)
- `pgbench` (CPU, RAM, disk)
- `Blender-benchmark` (GPU)

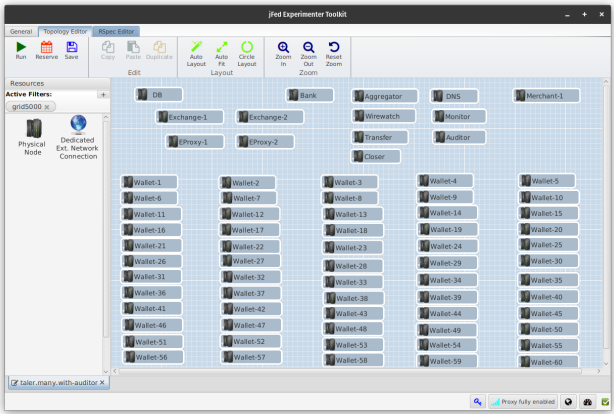
# Grid’5000 [1]

- Large-scale flexible testbed
- 800 nodes with total 15’000 cores
- Bare metal deployments
- Fully customizable software stack



# Platform Access

## jFed - Java-based GUI and CLI



# Running an experiment

**1.**

**Build Image (Kameleon)**

**2.**

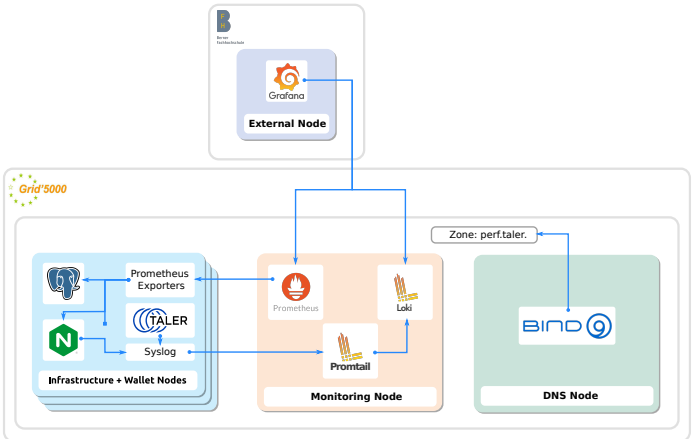
Grid'5000

**Copy Image to Grid'5000**

**3.**

**Allocate Experiment (jFed)**

# Experiment architecture



# Performance

## Various payment systems

| Bitcoin | PayPal | Visa |
| --- | --- | --- |
| 4 TPS | 193 TPS | 1’667 TPS |
| e-Krona [5] (Sweden) | e-CNY [4] (China) | Project Hamilton [2] (MIT) |
| 100 TPS | 10’000 TPS | 1’700’000 TPS |

**Know your requirements when benchmarking!**

# Performance: Future Work

Open issues:

- Compile SQL stored procedures to C
- Create realistic Taler benchmarks
- Systematic performance evaluation of all endpoints
- Best-available cryptographic primitive evaluation
- CBOR-support in the backend
- Real-time auditor parallelization

# References I


Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367, pages 3–20. Springer International Publishing, 2013.

# References II


Jim Cunha, Robert Bench, James Lovejoy, Cory Fields, Madars Virza, Tyler Frederick, David Urness, Kevin Karwaski, Anders Brownworth, Neha Narula.
Project hamilton phase 1 a high performance payment processing system designed for central bank digital currencies.
Technical report, Federal Reserve Bank of Boston and Massachusetts Institute of Technology Digital Currency Initiative, Feb 2022.
Available at `https://www.bostonfed.org/-/media/Documents/Project-Hamilton/Project-Hamilton-Phase-1-Whitepaper.pdf` [05.05.2022].

# References III


Ananya Kunar.
A report card on china's central bank digital currency: the e-cny, Jan 2022.
Available at https://www.atlanticcouncil.org/blogs/econographics/a-report-card-on-chinas-central-bank-digital-currency-the-e-cny/ [05.05.2022].

# References IV


People's Bank of China.
Progress of research & development of e-cny in china.
Technical report, People's Bank of China, Jul 2021.
Available at http://www.pbc.gov.cn/en/3688110/3688172/4157443/4293696/2021071614584691871.pdf [05.05.2022].

Sveriges Riskbank.
e-krona pilot phase 2.
Technical report, Sveriges Riskbank, Apr 2022.
Available at https://www.riksbank.se/globalassets/media/rapporter/e-krona/2022/e-krona-pilot-phase-2.pdf [05.05.2022].

# Acknowledgements

Co-funded by the European Union (Project 101135475).

Co-funded by
the European Union

Co-funded by SERI (HEU-Projekt 101135475-TALER).

**Project funded by**

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
**State Secretariat for Education,**
**Research and Innovation SERI**

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union.

Neither the European Union nor the granting authority can be held responsible for them.