A Comparative Performance Study of the Matrix `/sync` Endpoint on Synapse and Tuwunel

Windson Viana[1], Francisco Airton Silva[2], Francisco A. A. Gomes[1], Michel Sales[1], Roberto Ivo[1], Fernando A. M. Trinta[1], Rossana M. de C. Andrade[1], José A. F. de Macêdo[1], Vinícius Lagrota[3], Rodrigo Pacheco[3] and Paulo A. L. Rego[1],

[1]Universidade Federal do Ceará (UFC)

[2]Universidade Federal do Piauí (UFPI)

[3]Centro de Pesquisa e Desenvolvimento para a Segurança das Comunicações (CEPESC)

`{windson,michelsb}@ufc.br`

`{fernando.trinta,jose.macedo,rossana, paulo}@dc.ufc.br`

`robertoivo@insightlab.ufc.br, almada@crateus.ufc.br`

`faps@ufpi.edu.br {vinicius.1232,rodrigo.6874}@abin.gov.br`

Abstract. *Matrix is an open protocol for federated communication, primarily implemented by the Synapse server, with Tuwunel as an alternative. Client–server interaction relies on the synchronization (sync) mechanism, which delivers events and updates throughout the application lifecycle. This paper presents a controlled experimental comparison of Synapse and Tuwunel based on nearly 17,000 sync requests under increasing load and different filtering strategies. Results show that Tuwunel achieves lower response times in three of four configurations, but it does not yet support Simplified Sliding Sync (SSS) correctly. SSS on Synapse delivers the lowest latency across the evaluated settings. We also report saturation and resource trends under increasing load.*

1. Introduction

Matrix[1] is an open protocol for federated communication, widely adopted in distributed messaging systems and increasingly leveraged by governmental initiatives seeking communication channels with greater autonomy, interoperability, and control over data and infrastructure [Martins et al. 2026]. It's the foundation for several large-scale applications, including Tchap[2], used by the French government, with over 600,000 users, the BwMessenger[3] in Germany, and msg gov[4], an official messaging solution developed by the Brazilian government, demonstrating its suitability for sovereign and secure communication environments. Matrix also underpins commercial clients such as Element and Element X[5], which together support millions of users worldwide [Martins et al. 2025].

Synapse[6] is its most widely used server implementation. In parallel, Tuwunel[7] has

[1]https://matrix.org/
[2]https://tchap.numerique.gouv.fr/
[3]https://tinyurl.com/bwmessenger
[4]https://tinyurl.com/newsmsggov
[5]https://element.io/
[6]https://github.com/matrix-org/synapse
[7]https://github.com/matrix-construct/tuwunel

emerged, proposing distinct architecture with the goal of simplifying operation and reducing execution costs. Although both implement the same protocol, differences in design, component organization, and processing strategies can lead to distinct behaviors under load not yet well documented. At the center of the interaction between Matrix clients and servers is the `/sync` endpoint, responsible for delivering events, state, and updates needed to maintain a consistent view of rooms. Due to its reliance on polling mechanisms, this mechanism (*sync*) is the most frequently accessed endpoint on servers, in some cases accounting for more than 75% of all requests. This mechanism directly affects the volume of data transferred, the response time perceived by the client, and the consumption of resources on the server. Several synchronization strategies have been proposed in Matrix, including the use of filtering, deferred data delivery, and, more recently, pagination of the number of synchronized rooms, i.e., the Simplified Sliding Sync (SSS)[8].

Despite the relevance of *sync* for the user experience and for the operational cost of a Matrix server, there is still a lack of experimental characterization that directly connects the behavior observed on the client side (response time and volume of returned data) to the load imposed on the server (CPU and memory usage), especially when considering the adoption of more recent mechanisms such as SSS. Furthermore, it is not clear to what extent Tuwunel differences are reflected in saturation patterns, payload growth, and resource consumption as the amount of data to be synchronized increases.

This work presents a controlled experimental evaluation of the Matrix synchronization mechanism, comparing Synapse and Tuwunel under equivalent conditions and with a progressively increasing load of data to be synchronized. The proposal is to analyze how synchronization behaves as the volume of information returned to the client grows and as the computational effort required from the server increases, highlighting behavioral differences between the implementations. In this way, the article provides quantitative evidence that supports both the choice of server and the understanding of the cost of *sync* in intensive-use scenarios. This study is part of the msg gov project [9] and reflects real operational requirements rather than a purely external benchmark.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 details the *sync* in Matrix, establishing the concepts needed to interpret the results. Section 4 describes the architecture explored and the components relevant for comparing servers. Sections 5 and 6 present the experimentation methodology and its results. Finally, Section 7 concludes the paper with the main findings and future directions.

2. Related Work

Several studies were identified as aligned with this paper's goals [Martins et al. 2026]. Three of them analyze the distribution of users across rooms and servers in the Matrix ecosystem or fault tolerance [Jacob et al. 2019, Jacob et al. 2021, Stocker et al. 2025], contributing to an understanding of how the platform behaves as the number of users, rooms, and events increases. Jacob et al. [Jacob et al. 2019] studied the scalability of Matrix as a message-oriented synchronization middleware in federated platforms, discussing how synchronization and event volume can put pressure on the server and affect response times in scenarios close to real-world usage. In [Jacob et al. 2021], the authors investigate the Matrix Event Graph (MEG) as a replicated data type for decentralized apps, validating

[8]https://github.com/matrix-org/sliding-sync
[9]https://tinyurl.com/msggovapp

it as a Conflict-free Replicated Data Type (CRDT) with strong eventual consistency. The work helps contextualize costs associated with replication and concurrency.

[Stocker et al. 2025] investigated Synapse's behavior under crash-type failures, in which the process stops responding. The proposal evaluated to what extent a user's server downtime affects their communication and whether the protocol is able to recover undelivered messages after the server comes back online. Three other works emphasize security and privacy in Matrix [Müller et al. 2021, Albrecht et al. 2023]. Muller et al. [Müller et al. 2021] dealt with the incorporation of informal communications into digital industrial management systems; although it is not a Matrix-centered study, it reinforces the presence of messaging platforms in operational workflows and the need to treat them as part of the support infrastructure. Albrecht et al. [Albrecht et al. 2023] showed vulnerabilities in the Matrix protocol and in the Element client, highlighting the effects of malicious servers on confidentiality and authentication guarantees; this type of analysis is relevant because security requirements and implementation choices can affect critical routes and, consequently, the performance perceived by the user.

Unlike previous studies, which tend to emphasize dimensions such as the structural scalability of the Matrix ecosystem or security and privacy concerns, this work focuses on a gap of an operational nature: how the synchronization mechanism behaves as data volume and concurrency increase. The focus is on the `/sync` endpoint and its more recent variant, SSS, as they are central components of client–server interaction and directly influence perceived latency, the size of the returned payload, and the computational cost on the server. In this sense, the comparison between Synapse and Tuwunel is treated as part of the problem, since implementation differences can produce distinct responses under the same load conditions.

3. *Sync* Matrix Function

In this work, we focus on the synchronization feature: the **sync** endpoint [10]. It is the main way in which Matrix clients receive from the homeserver the events necessary to keep the local state updated, including new messages, room state changes, and other notifications associated with the user session. Figure 1 illustrates the basic flow of this behavior: Client A sends a message ("Hello B, how are you?") to a room shared with Client B, and the homeserver registers this event. Then, Client B does not "receive" the message via a direct push from the server; instead, it executes a pooling *sync* request to the homeserver, and the *sync* response contains the newly created event, allowing the client to update its timeline and/or notifications.

The *sync* feature acts as the event delivery mechanism for the client, and the operating cost increases as active users, rooms, and accumulated events between synchronizations grow. Because *sync* is triggered both periodically (e.g., every 30 seconds) and in response to specific events, such as after login, when entering rooms, and during user interactions, it tends to concentrate a large portion of the client–server traffic; becoming crucial for perceived latency and overall server load. From an implementation standpoint, *sync* on homeserver assembles a JSON response aggregating events from multiple rooms linked to the user. The volume and cost of this response vary with factors such as: number of rooms, number of pending events per room, limits on returned messages, applied filters, and inclusion of additional data (e.g., member list). For example, if between the

[10] `https://spec.matrix.org/v1.17/client-server-api/#syncing`

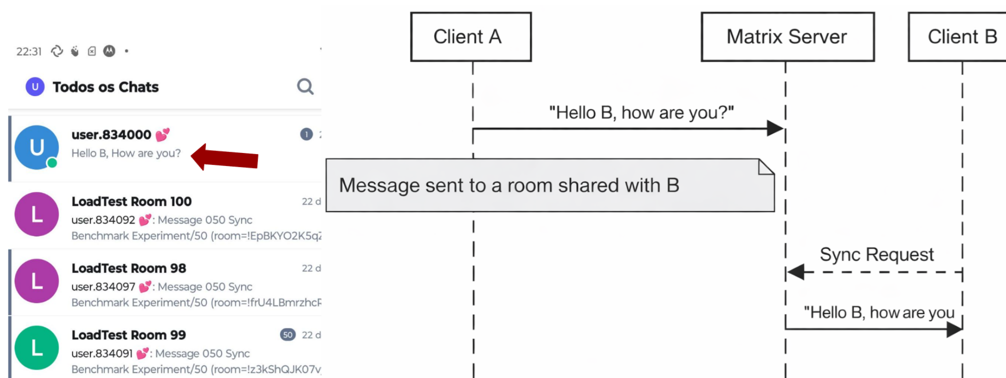


Figure 1. Basic functionality of *sync*

last and the current *sync*, the user received N messages across M rooms, the server will decide, depending on the *sync* strategy, which of these N messages will be sent. For instance, the server may choose to send only the most recent message received from each of the rooms the user is a member of. Under high load, this process puts pressure on CPU, memory, and database, as each *sync* involves queries and the assembly of large structures, and differences in sync strategy can directly alter the response time and server stability.

3.1. Filters Timeline Limit and Lazy-Loading Room Members

In the Matrix protocol, the `/sync` endpoint provides a filtering mechanism that allows clients to precisely control the information returned during synchronization, limiting transferred data volume, reducing JSON response size, and adjusting server-side processing costs. For instance, the `timeline.limit` filter defines the maximum number of events returned per room, where smaller values reduce payload size and serialization overhead at the cost of requiring additional room-specific requests, while larger values increase transferred history and server load.

The `state_lazy_load_members` filter defines whether the state information related to room members should be loaded lazily or completely. When set to **true**, the server avoids including all member state events in the initial response, returning only those strictly necessary, which reduces the volume of data in rooms with many participants. When set to **false**, the complete member state is sent, increasing processing and transfer costs, but providing a more comprehensive view of the room state in each sync.

The chosen synchronization strategy directly affects the number of missing messages/events that must be fetched when a user enters a specific conversation. If the synchronization process returns only a single message per room, entering a room requires loading a larger number of additional messages and events (e.g., a User C is a new member of the room). For example, if only one message is retrieved during synchronization, the client must request the remaining 49 messages upon room entry. In contrast, strategies that return a larger portion of each room's timeline, such as those used by clients like Element Web, may retrieve 20 messages per room during synchronization. In this case, entering a specific room requires fetching only the remaining 30 messages, thereby reducing both latency and server workload at the moment of room access.

The combination of `timeline.limit` and `state_lazy_load_members` allows exploring different synchronization strategies, ranging from minimal and highly

restricted responses to more complete and costly synchronizations. These selected combinations emulate configuration choices available in clients such as Element Classic, Element Web, and Fluffy. By testing these combinations on both Synapse and Tuwunel servers, it becomes possible to compare how each implementation handles the progressive increase in data load, identifying differences in scalability, response time, and how filters influence the cost of syncing in intensive usage scenarios.

3.2. Simplified Sliding Sync - SSS

SSS is an evolution of the traditional Matrix sync mechanism that seeks to more precisely control the volume of data exchanged between client and server as the number of rooms grows. Instead of uniformly synchronizing all rooms, SSS introduces the concept of a sliding window, in which the server defines which subsets of rooms should be synchronized in each request. This approach follows the same principle as the `/sync` filters, that is, limiting transferred data and processing cost, but applied directly to room selection. Instead of synchronizing all rooms at once, the client initially requests a subset of the room list, typically limited to the first 20 rooms, with at most **one message per room** included in the response. As interaction progresses, additional synchronization windows are requested, gradually expanding the synchronized set to larger ranges, commonly increasing to 100 and then 200 rooms, until full coverage of the room list is achieved.

From a strategy perspective, SSS shifts synchronization from a bulk, room-complete model to an incremental, window-based approach. This design reduces the size of individual responses and distributes both network and server-side processing costs over time. A drawback of this strategy is that it increases the number of synchronization requests and requires server-side ordering logic to determine which rooms should be returned in each window. SSS is specified in the Matrix Client–Server API[11] and, at the time of writing, remains a beta feature, with ongoing refinement and partial adoption across Matrix clients. Another drawback of SSS is that it is possible for a user to enter a room for which no messages have been retrieved during synchronization. In such scenarios, when the user clicks to open the room, the client triggers the retrieval of all missing messages and events.

4. Infrastructure and Implementation of the Assessment Environment

Figure 2 shows the test scenario. The experiments were conducted in a local network environment in which one machine was dedicated to simulating clients issuing synchronization and room access requests, while a separate machine was used to host the Synapse and Tuwunel servers. Synapse is the reference Matrix homeserver implementation, written in Python, and is widely adopted in production deployments. In our setup, Synapse was backed by a PostgreSQL database, which is the recommended configuration for production environments and supports high concurrency and persistence requirements. Tuwunel is an alternative Matrix server implementation written in Rust, designed with a focus on performance and resource efficiency. In our experiments, Tuwunel used RocksDB as its database backend, following the default installation configuration. This database, unlike PostgreSQL, is a key-value store focused on high read performance. However, it has limitations in terms of query capabilities and index usage. Tuwunel is tightly coupled to this implementation. In this sense, our experiments are comparing not only the servers, but

[11] https://tinyurl.com/sssmatrix

rather the server–storage pairs (Synapse, Tuwunel) and their standard storage solutions (PostgreSQL, RocksDB).

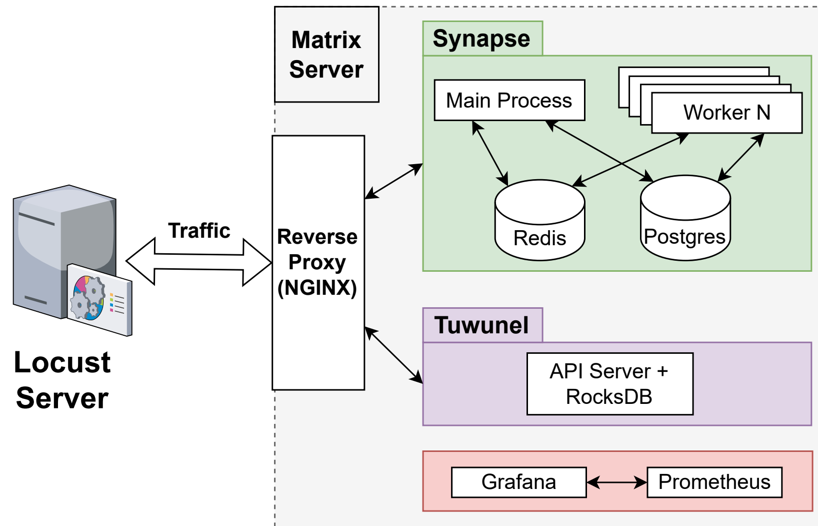


Figure 2. Matrix architecture used in the test environment.

The server machine used an Intel(R) Core(TM) i9-14900K processor with 24 cores, 32 threads, and 192 GB of main memory. The experimental setup deployed both Tuwunel and Synapse on this machine and ran Ubuntu Server 24.04 LTS as the operating system. This hardware and software configuration provided substantial computational capacity and memory resources. The setup executed both servers inside virtualized container environments with different configuration setups. Based on prior studies, the setup adopted a high-scalability architecture for Synapse, composed of the Main Process and multiple workers. In contrast, we adopted a monolithic configuration for Tuwunel, following the default installation instructions. In addition, each architectural component illustrated as a white box in Figure 2 ran as an individual container.

In our experiment, Synapse was deployed using a modular and scalable configuration inspired by real production environments. At its core, the Main Process acts as a coordination unit, responsible for routing client requests, managing federated communication, and orchestrating interactions among worker services. It maintains continuous communication with the data layer and distributed services, enabling efficient task delegation under high demand. To maximize CPU utilization, this experimental setup distributes responsibilities across 18 specialized workers. Examples include background workers for asynchronous tasks, media repository workers for handling media content, federation sender workers for inter server communication, and generic workers that support client requests and event processing (e.g., event creation and client reading).

To enable observation of system behavior during experiments, Prometheus collects real-time metrics from Synapse and Tuwunel services and the container engine itself. Metrics such as CPU and memory usage, request throughput, and response latency are collected in detail, allowing a comprehensive view of system behavior. This information is presented through Grafana, which offers dynamic dashboards for monitoring system health, identifying bottlenecks, and supporting performance analysis.

5. Experiment Methodology

5.1. Research Questions

RQ1: What is the impact of the `timeline.limit` parameter and the `state_lazy_load_members` filter on the performance of the initial sync? This research question investigates how different synchronization configurations affect both client-side latency and server-side resource consumption.

RQ2: What are the performance differences between Synapse and Tuwunel for synchronization operations?

The Matrix protocol is implemented by multiple server implementations with distinct architectural choices that may influence overall performance.

RQ3: What are the performance differences between classic synchronization strategies and Simplified Sliding Sync?

Simplified Sliding Sync aims to reduce the size of synchronization responses by limiting the number of rooms and messages returned per request; however, its impact on server-side performance and the extent to which it improves client-side response times are not yet well understood.

5.2. Rooms and Number of Users

Two user groups were created to populate the Synapse and Tuwunel databases. The **first group** was composed of 100 users, each belonging to 100 rooms, where each room contains all 100 members. The **second group** was composed of 300 users, each belonging to 300 rooms, where each room contains 300 members. In both groups, each room was populated with 50 messages. The user responsible for writing the 50 messages was randomly selected during scenario generation. 50 was chosen because it corresponds to the default pagination behavior of the Element Classic client, which loads the most recent 50 messages when entering a room. Messages follow the template:

`Message ID Sync Benchmark Experiment/TOTAL`

This fixed template ensures message content remains consistent across runs, so differences in payload size are driven primarily by synchronization strategy and state inclusion rather than by message variability. The **first group** of users was designed to simulate heavy users who participate in multiple work-related groups, where rooms typically have a moderate number of members, but the user is involved in many parallel conversations. The **second group** aims to represent heavy users engaged in large groups with a high number of participants, reflecting environments with high fan-out and increased synchronization complexity. It is worth noting that real-world deployments can exhibit even larger groups; for instance, the official Matrix room on Element has more than 5,000 members.

5.3. User Simulation

A Python-based Locust workload [12] was used to simulate users accessing the server in parallel while employing the same synchronization strategy under evaluation. The following scenarios were defined:

[12]Locust is an open-source, Python-based load testing framework that enables the simulation of concurrent users and the definition of user behavior through programmable scenarios, making it suitable for evaluating the performance and scalability of web services.

- **C1:** A single user performs an initial synchronization and requests the messages of a room. The user can be of both groups.
- **C100:** 100 users of the first group perform an initial synchronization and request the messages of a room.
- **C200:** 200 users of the second group perform an initial synchronization and request the messages of a room.

In all three cases, when entering a specific room, clients request the set of messages necessary to complete the full message history. In this sense, strategies that prioritize sending fewer messages during the initial sync will need to request more messages when entering a specific room.

5.4. Sync strategies

Beyond the use of SSS, the experiments explore four distinct configurations of *sync*, obtained from the combination of two filters used by the protocol: `timeline.limit`, which controls the number of events returned per room, and `state_lazy_load_members`, which defines the state loading strategy for members. These configurations are compactly identified throughout the graphs by labels such as 1T, 5T, 10F, and 20T (see Table 1 for more details), allowing comparison of scenarios with different timeline depths and state loading strategies. 10F corresponds to the Element Classic configuration by default, and 20T the Element Web configuration.

Table 1. Filter settings used in the *sync* experiments.

Config.	timeline.limit	lazy load members	Description
1T	1	true	Returns only the most recent event per room, using lazy loading for member state.
5T	5	true	Returns up to five events per room, while maintaining lazy loading of member state.
10F	10	false	Returns up to ten events per room, including the full member state in the synchronization.
20T	20	true	Returns up to twenty events per room, using lazy loading for member state.

5.5. Preliminary tests

For each scenario and server, preliminary tests were conducted to verify correctness and execution conditions. First, these tests revealed that a scenario with 300 users of the second group (with 300 rooms) generated a large number of failures on Synapse, which led us to evaluate only 200 users of this group. Additionally, SSS did not complete on Tuwunel in the 300-room dataset. We reproduced this behavior using Element X connected to Tuwunel, where synchronization consistently stalled after the first window, effectively limiting correct synchronization to the initial 20 rooms. In the C200 scenario, SSS failed to complete on Synapse due to limitations in the default SSS configuration, with most requests returning 504 Gateway Time out. After enabling a worker based configuration for SSS, synchronization completed correctly.

5.6. Procedure

We started Locust with a randomly selected combination of scenario and sync strategy (e.g., 1T with 200 users from the second group) and executed separately for each server. A report was generated for each execution, containing server response times, response JSON sizes, and CPU and memory utilization metrics. Locust was then restarted for the

next scenario and synchronization strategy. Each configuration was executed five times, resulting in a total of 110 executions. 6,000 synchronization requests were issued for each server, covering the four filter combinations, in addition to 5,000 SSS requests executed exclusively on Synapse. Execution times ranged from 12 seconds for scenario C100 with the 1T strategy to approximately 13 minutes for the 10F strategy on Tuwunel. For scenario C1, due to short response times, each configuration was executed 10 times on each server.

6. Results

6.1. RQ1: What is the impact of the `timeline.limit` parameter and the `state_lazy_load_members` filter on the performance of the initial sync?

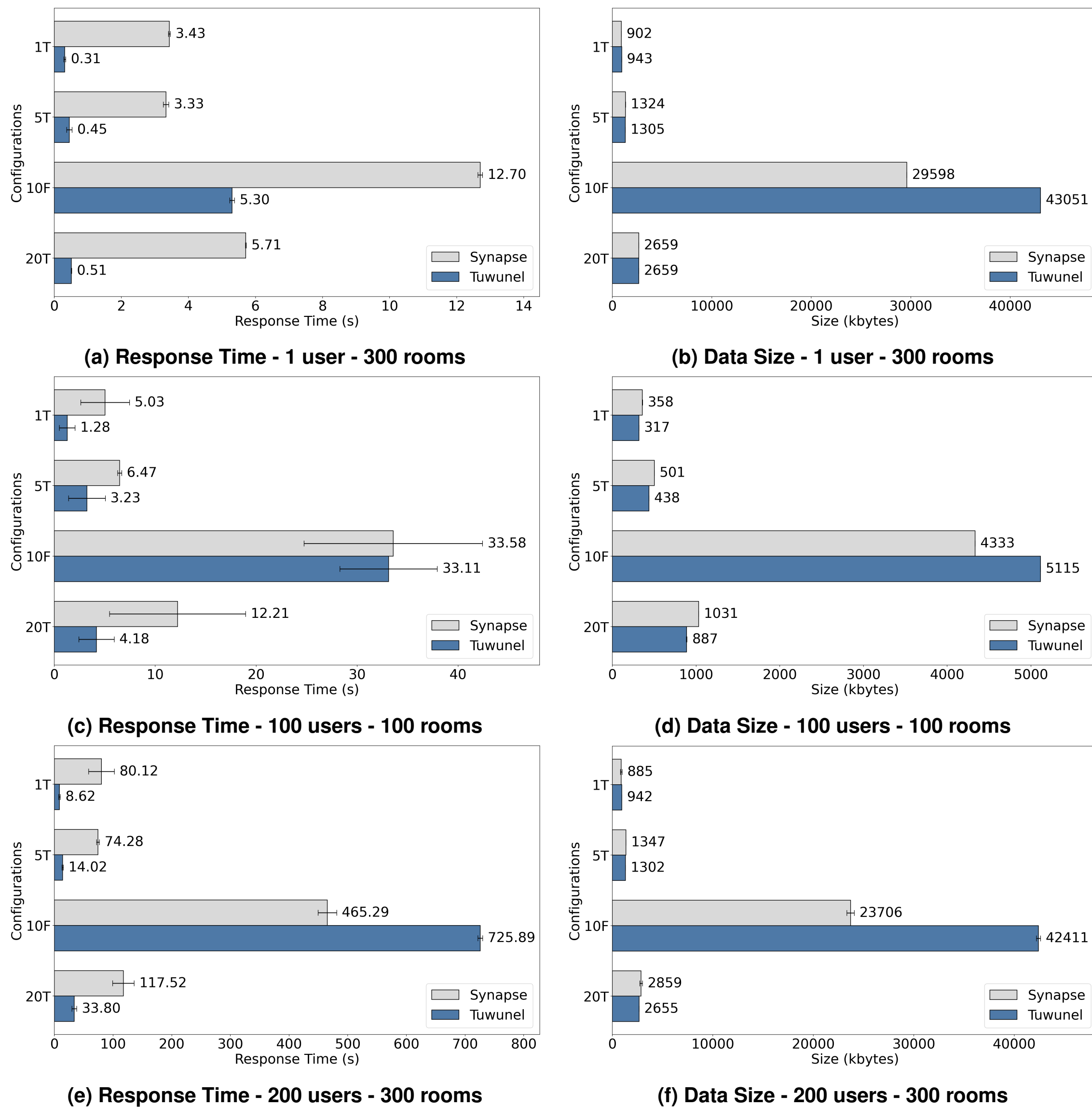
Main result: The use of the `state_lazy_load_members` filter has the most impact on the sync strategies, with configuration 10F exhibiting the poorest performance. By contrast, limiting the number of messages has a comparatively smaller impact; however, it still yields measurable effects on server response time.

The six graphs in Figure 3 organize these results into three load scenarios, simultaneously varying the number of users and the number of rooms associated with each execution. In particular, the first two graphs consider the scenario with 1 user and 300 rooms (C1), the two central graphs represent the scenario with 100 users and 100 rooms (C100), and the last two correspond to the scenario with 200 users and 300 rooms (C200).

Figures 3a and 3b present the scenario with 1 user and 300 rooms, allowing for isolated observation of the impact of the synchronized context volume on *sync*, without the interference of concurrency between multiple clients. Graph 3a shows the response time, in which Synapse shows a noticeable increase as the `timeline.limit` grows, with the 10F configuration standing out, concentrating the longest response time among the four configurations evaluated. **Tuwunel maintains significantly shorter times** in all configurations, although it also shows variation according to the timeline depth. Figure 3b, which shows the amount of synchronized data, shows that the size of the responses increases sharply when the limit of events per room increases, again with the 10F configuration standing out as the most costly in terms of data volume. In this case, Tuwunel returns larger responses than Synapse, indicating differences in how each implementation serializes and organizes the sync data when the lazy loading members filter is not used.

In Figure 3c and 3d, the scenario with 100 users and 100 rooms is analyzed, in which the *sync* simultaneously reflects the effect of concurrency between clients and the volume of data associated with each room. In Figure 3c, which shows the response time, it can be observed that all configurations show an increase compared to the scenario with only one user, with emphasis on configuration 10F, which concentrates the longest times in both Synapse and Tuwunel. In Figure 3d, referring to the amount of synchronized data, the behavior varies more subtly between the implementations. Configuration 10F again stands out as the most costly in terms of volume, with higher response rates in Tuwunel. In contrast, in configuration 20T, Synapse returns a higher volume of data than Tuwunel, indicating that, even with lazy loading enabled, increasing the event limit per room is sufficient to reverse the pattern observed in other configurations.

In Figures 3e and 3f, the most heavily loaded scenario is considered, with 200 users and 300 rooms, in which the *sync* starts operating under high concurrency and a large volume of synchronized context. In Figure 3e, which shows the response time,



(a) Response Time - 1 user - 300 rooms **(b) Data Size - 1 user - 300 rooms**

(c) Response Time - 100 users - 100 rooms **(d) Data Size - 100 users - 100 rooms**

(e) Response Time - 200 users - 300 rooms **(f) Data Size - 200 users - 300 rooms**

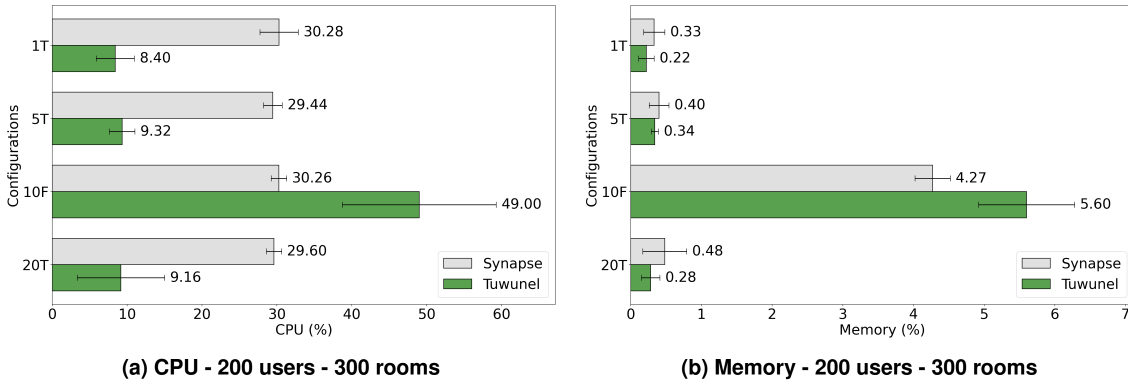
Figure 3. Synapse versus Tuwunel focusing on response time and data size

a significant increase is observed in all configurations, especially in configuration 10F, which concentrates the highest values. It is worth noting that, in all executions of the 10F strategy on Synapse, approximately 1% of the requests resulted in failures due to timeouts. Timeouts were recorded as failures by Locust and excluded from average response-time calculations. Unlike the scenario with 100 users, Synapse shows a lower response time than Tuwunel in 10F, indicating a reversal in the relative behavior of the implementations under higher load pressure. Configurations 1T and 5T also show a substantial increase in latency compared to the previous scenarios, but maintain the same order of magnitude between servers, while 20T appears again at an intermediate level.

In Figure 3f, referring to the amount of synchronized data, the behavior between Synapse and Tuwunel becomes closer in several configurations. The 10F configuration remains the most costly in terms of volume, with Tuwunel returning larger responses, reinforcing the impact of full member state loading in high concurrency scenarios.

6.2. RQ2: What are the performance differences between Synapse and Tuwunel for synchronization operations?

Main result: Tuwunel exhibited lower response times than Synapse in scenarios that employed the `state_lazy_load_members` filter. While both servers showed minimal variation in memory consumption, an increase in CPU usage was observed. In the C200 scenario, Synapse consumed approximately three times more CPU resources than Tuwunel, with the exception of configuration 10F.



(a) CPU - 200 users - 300 rooms **(b) Memory - 200 users - 300 rooms**

Figure 4. CPU and Memory variation

Figure 4a reports the average CPU consumption variation for Synapse and Tuwunel. Overall, CPU increase remains consistently high for both servers, reinforcing that *sync* processing represents a substantial workload on the backend. The 10F configuration, which concentrates a larger volume of synchronized data in each request, triggers a pronounced rise in CPU usage on Tuwunel, suggesting a stronger sensitivity of this implementation to configuration changes and heavier synchronization workloads. In contrast, Synapse exhibits relatively stable CPU increase across the evaluated configurations, indicating that, within the tested range, increasing the synchronized-data volume does not translate into a proportional increase in CPU demand on this server.

Figure 3b presents the average increase of memory consumption during the same experiments. Compared to CPU, memory usage stays low across all scenarios, pointing to limited memory pressure even when synchronization traffic grows. Still, a clearer increase appears again in the 10F configuration for both servers, consistent with the larger amount of state and events processed per *sync* response. Even in this heavier configuration, memory consumption increases modestly around 4% for Synapse and 5% for Tuwunel, indicating that the *sync* mechanism is predominantly CPU-bound in these settings, while its impact on memory remains comparatively small.

6.3. RQ3: What are the performance differences between classic synchronization strategies and Simplified Sliding Sync?

The results indicate that SSS generally reduces the amount of data transferred and delivers the shorter response times under higher concurrency.

Figure 5 presents a performance analysis of Synapse when using the traditional *sync* endpoint and the SSS one, considering both response time and synchronized data size under different workloads. By pairing latency and payload volume, the figure enables a

direct interpretation of how synchronization strategies impact user-perceived performance and server-side costs as the number of rooms and users increases.

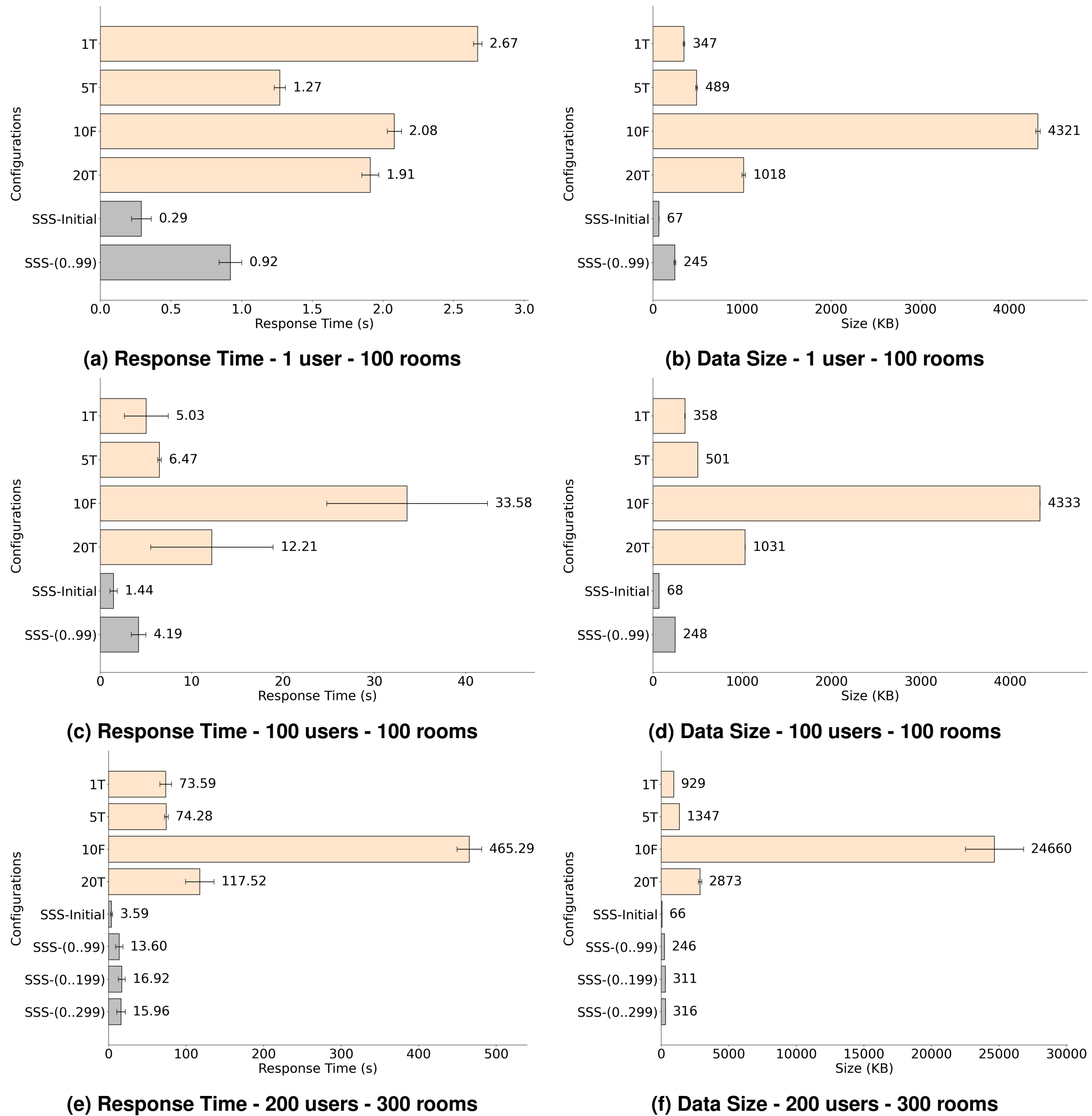


Figure 5. SSS detailed performance evaluation.

Figures 5(a)–(f) show that, across all evaluated scenarios, each Simplified Sliding Sync (SSS) request exhibits substantially lower response times when compared to traditional *sync* configurations. This behavior is consistent even under higher concurrency levels, as illustrated in Figure 5(c), where traditional *sync* configurations with lazy-loaded members achieve lower response times only when considering a single, monolithic request. In contrast, SSS decomposes synchronization into multiple lightweight requests, effectively distributing and spacing the load on the client side. While this approach may increase the total time required for all messages to be delivered, it significantly improves responsiveness at each synchronization step, reducing perceived latency and enabling smoother client interaction under load.

In our server configuration, the adoption of SSS does not introduce a substan-

tial increase in resource consumption when compared to other synchronization strategies. CPU usage followed trends similar to the traditional configurations, with an observed increase of approximately 30% under higher concurrency, while memory consumption remained largely stable, varying by around 0.3%. These results indicate that, despite the higher number of synchronization requests issued by SSS, its incremental and windowed design avoids additional pressure on server resources, maintaining a resource profile comparable to that of conventional *sync* approaches. It is worth noting that if we sum all SSS requests, we approach the 1T option, with the advantage that the UI can already display part of the data to the user. The downside is if the rooms of interest to the user only appear in the final SSS requests, which would force the user either to wait or to request the room's messages upon clicking, thereby breaking the initial sync logic.

7. Final Considerations

This work presented an experimental evaluation of the *sync* mechanism in the Matrix protocol, comparing Synapse and Tuwunel under varying workloads (number of users and rooms) and synchronization strategies. The analysis encompassed response time, synchronized data volume, server-side CPU and memory consumption, as well as a characterization of the SSS mechanism.

The results show that the cost of *sync* increases with higher data volumes and concurrency levels. The 10F configuration yields the highest response times and response sizes. Overall, Tuwunel achieved lower response times in less loaded scenarios and exhibited lower variation in CPU usage across configurations, although it showed increased sensitivity as the load intensified. Synapse, in contrast, maintained consistently stable CPU consumption across configurations. For both servers, memory consumption exhibited limited variation, even under the most demanding scenarios.

The SSS evaluation revealed an incremental, window-based behavior that distributes synchronization costs over time by decomposing sync into lightweight requests. While this approach may increase the overall time required for all messages to be delivered under high concurrency, each individual request exhibits significantly lower response times. Despite the higher number of sync requests, no additional resource consumption was observed, highlighting trade-offs between per-request responsiveness and total synchronization completion time for homeserver selection and sync configuration in intensive usage scenarios. It is important to note, however, that these results were only achievable due to the adoption of a specific distribution configuration, as current documentation does not clearly describe how this configuration should be optimally applied. Once a correct and well-documented implementation becomes available in Tuwunel, the expectation is that its performance will further improve, potentially surpassing Synapse by following the performance trends observed in other synchronization strategies.

Some limitations of this study should be acknowledged. We simulated heavy users under specific configurations, which may not fully represent the diversity of usage profiles observed in messaging applications deployed in governmental services. We compared the configurations recommended by the Synapse and Tuwunel teams. There is an expectation that Tuwunel's performance can be further improved by increasing the number of workers used. However, the lack of indexing in the database may become detrimental as the number of users and the volume of data on the servers grow significantly. Measuring variations in memory and CPU consumption is inherently challenging, as other system services or operating system functionalities may execute concurrently and interfere with resource us-

age. Despite these limitations, repeated experimental runs consistently exhibited the same behavioral patterns across synchronization strategies and server implementations, thereby strengthening the support for the conclusions presented.

As future work, we plan to investigate the impact of synchronization algorithms on mobile and web clients in terms of processing time and resource consumption, as well as to propose strategies to reduce the size of the JSON-based payload.

References

Albrecht, M. R., Celi, S., Dowling, B., and Jones, D. (2023). Practically-exploitable cryptographic vulnerabilities in matrix. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 164–181. IEEE.

Jacob, F., Beer, C., Henze, N., and Hartenstein, H. (2021). Analysis of the matrix event graph replicated data type. *IEEE access*, 9:28317–28333.

Jacob, F., Grashöfer, J., and Hartenstein, H. (2019). A glimpse of the matrix (extended version): Scalability issues of a new message-oriented data synchronization middleware. *arXiv preprint arXiv:1910.06295*.

Martins, J. A., Rego, P. A., de Macêdo, J. A., Silva, F. A., and Lagrota, V. (2026). Matrix protocol: a comprehensive systematic mapping study. *Journal of Cloud Computing*, 15(1):20.

Martins, J. A. P., Rego, P. A. L., Macêdo, J. A. F. d., Andrade, R. M. C., Bonfim, M. S., Ivo, R. F., Costa, V. L. R. d., Pacheco, R. P., and Silva, F. A. P. d. (2025). Protocolo matrix: Conceitos, arquitetura, aplicações e desafios. In *Jornada de Atualização em Informática 2025*. SBC, Porto Alegre, RS, Brasil.

Müller, M., Lee, J.-U., Frick, N., Stangier, L., Gurevych, I., and Metternich, J. (2021). Extracting problem related entities from production chats to enhance the data base for assistance functions on the shop floor. *Procedia CIRP*, 103:231–236.

Stocker, V., de Camargo, E. T., and De Bona, L. C. (2025). Uma investigação sobre tolerância a falhas no servidor de comunicação synapse. In *Workshop de Testes e Tolerância a Falhas (WTF)*, pages 57–70. SBC.