BANCA D'ITALIA

EUROSISTEMA

Mercati, infrastrutture, sistemi di pagamento

(Markets, Infrastructures, Payment Systems)

Real-Time Gross Settlement systems: breaking the wall of scalability and high availability

by Mauro Arcese, Domenico Di Giulio and Vitangelo Lasorella

March 2021

Number 2

BANCA D'ITALIA
EUROSISTEMA

Mercati, infrastrutture, sistemi di pagamento
(Markets, Infrastructures, Payment Systems)

Approfondimenti
(Research Papers)

Real-Time Gross Settlement systems: breaking the wall of scalability and high availability

by Mauro Arcese, Domenico Di Giulio and Vitangelo Lasorella

Number 2 – March 2021

The papers published in the 'Markets, Infrastructures, Payment Systems' series provide information and analysis on aspects regarding the institutional duties of the Bank of Italy in relation to the monitoring of financial markets and payment systems and the development and management of the corresponding infrastructures in order to foster a better understanding of these issues and stimulate discussion among institutions, economic actors and citizens.

The views expressed in the papers are those of the authors and do not necessarily reflect those of the Bank of Italy.

The series is available online at www.bancaditalia.it.

Printed copies can be requested from the Paolo Baffi Library: richieste.pubblicazioni@bancaditalia.it.

ISSN 2724-6418 (online)
ISSN 2724-640X (print)

Banca d'Italia
Via Nazionale, 91 - 00184 Rome - Italy
+39 06 47921

Designed by the Printing and Publishing Division of the Bank of Italy

Real-Time Gross Settlement systems:
breaking the wall of scalability and high availability

by Mauro Arcese, Domenico Di Giulio and Vitangelo Lasorella*

Abstract

By leveraging the progress of information technology and computer networks, Real Time Gross Settlement systems (RTGS), which first appeared in the early 1990s, have quickly become the standard for the settlement of interbank payments in central bank money. However, the market of retail payments still relies on batch-based netting systems, as the huge volume of retail payments makes the architecture of a typical RTGS impractical in this context.

The Eurosystem's decision to provide a settlement service for instant payments has laid the foundation for the transformation of retail systems into real-time systems. A new design for settlement engines is needed, which should now be capable of settling an extremely high volume of payments in real time, while also operating continuously and exposing low transaction costs.

This need has been the main driver for the design of the TIPS (Target Instant Payment Settlement) engine, a distributed, event-driven architecture based on principles of reactive applications and running on Linux/x86 commodity systems for cost efficiency.

This paper shows how the traditional design of a mostly centralized, vertically scalable RTGS can be replaced by a distributed, horizontally scalable system, provided that the proper mix of parallel and sequential computation is used.

Keywords: TIPS, RTGS, Distributed systems.

Sintesi

Facendo leva sul progresso della tecnologia dell'informazione e delle reti di telecomunicazione, i sistemi di regolamento lordo in tempo reale (*Real Time Gross Settlement Systems* - RTGS), apparsi dall'inizio degli anni Novanta, sono rapidamente divenuti la soluzione standard per regolare in moneta di banca centrale i pagamenti interbancari. Il mercato dei pagamenti al dettaglio, per contro, continua a impiegare sistemi di compensazione, basati su elaborazioni *batch*: l'enorme volume dei pagamenti al dettaglio rende infatti l'architettura di un tipico sistema RTGS inadatta per tale ambito.

La decisione dell'Eurosistema di fornire un servizio di regolamento per i pagamenti istantanei ha posto le basi per la trasformazione dei sistemi al dettaglio in sistemi in tempo reale. È emersa la necessità di un nuovo disegno per i *motori* di regolamento, che ora dovrebbero essere in grado di regolare in tempo reale un volume estremamente elevato di pagamenti, funzionando in modo continuativo e con bassi costi.

Questa esigenza è stata il principale stimolo che ha portato alla progettazione dell'*engine* TIPS (Target Instant Payment Settlement), un'architettura distribuita ed *event-based* costruita sui principi di applicazioni *reactive*, e in grado di contenere i costi grazie all'impiego di sistemi *commodity* di tipo Linux / x86.

Il lavoro mostra come il design tradizionale di un RTGS, per lo più centralizzato e scalabile solo verticalmente, possa essere sostituito da un sistema distribuito e scalabile orizzontalmente, a condizione che venga utilizzato il giusto mix di calcolo parallelo e sequenziale.

* Bank of Italy, ITC Development Directorate.

CONTENTS

1 INTRODUCTION[1]

A Real-Time Gross Settlement system - RTGS (Committee on Payment and Settlement Systems of the central banks of the Group of Ten countries 1997), (Committee on Payment and Settlement Systems - Technical Committee of the International Organization of Securities Commissions 2012), is a centralized system, usually hosted by a central bank, which handles interbank payment requests submitted by the banking community served by that central bank. Large payments are settled in real time,[2] as they typically transfer huge amounts of money from one bank to the other, and there is a need to make sure that these amounts are immediately available on the accounts within the central bank's system (the RTGS itself). For this reason, an RTGS system is typically conceived as a huge centralized system whose purpose is mainly to provide the expected quality of service in terms of payment latency and service reliability.

Because of their architecture, which makes use of extremely powerful machines having an outstanding CPU and I/O capacity, the operating costs of such systems are typically very high; this means the corresponding transaction fees paid by commercial banks are high as well. This is deemed acceptable as the cost per transaction is relatively small if compared with the amount of a large payment. Costs would become lower if the same systems could also be used to process huge volumes as for retail payments, with positive effects on transaction fees. However, this is impractical, as even large centralized computers with an outstanding capacity cannot scale up to hundreds of millions of transactions per day, as expected by a retail payment system.

To overcome the aforementioned performance problem, the market of retail payments is mainly based on the 'netting' concept: multiple systems allow the banks to interact with each other, 'moving the information' around, but without really 'moving the money'. Many payments are batched together by netting systems, which send a single payment request to the RTGS to settle all these payments in one shot: what is settled is only the sum of all the payments batched, so that just one transaction executed in the RTGS is sufficient to complete thousands of retail payments. This way the cost of one single RTGS transaction is assigned to several retail payments, and the cost per payment becomes much lower. Yet this also means that, until the money is finally transferred within the RTGS, there is no guarantee that an underlying payment will be successful, and most of all, the money will not be available on the beneficiary side until then.

The goal of this paper is to analyze why an RTGS, as it is usually designed, cannot cope with the huge volumes generated by retail payments, and how to find a better way to design a more fit-for-purpose system. While processing hundreds of millions of transactions per day, a real-time system for retail payments must also be available with no service interruption (24x7x365), as individuals are *always* exchanging money, not just during the working hours of commercial banks. Obviously it would also be good to cut down transaction fees by leveraging on commodity computers, not just by raising volumes.

1.1 An RTGS for high volumes

Within a diagram that crosses speed and volume (see Figure 1), existing settlement infrastructures can be positioned in two areas:

- RTGS systems are really fast, but provide their services with low latency, which is the time needed for every payment request to be finalized, only for a limited volume: they hardly scale up, so their use is limited to large-value payments, where immediate processing matters;

1. We would like to thank Federica Sbarra for her assistance in the review of the text, Fabrizio Bersani, Pietro Tiberi, Costantino Colangelo, Paolo Libri, Michele Romanelli, Arturo Baldo, Giuseppe Zingrillo and Stefano Siviero for revising the article and for their useful comments. All errors are ours. *The views expressed in the paper are solely those of the authors and do not necessarily represent those of the Bank of Italy.*

2. The reasons why RTGS systems are essential for settling large-value funds transfer are explained in (Chakravorti 2000).

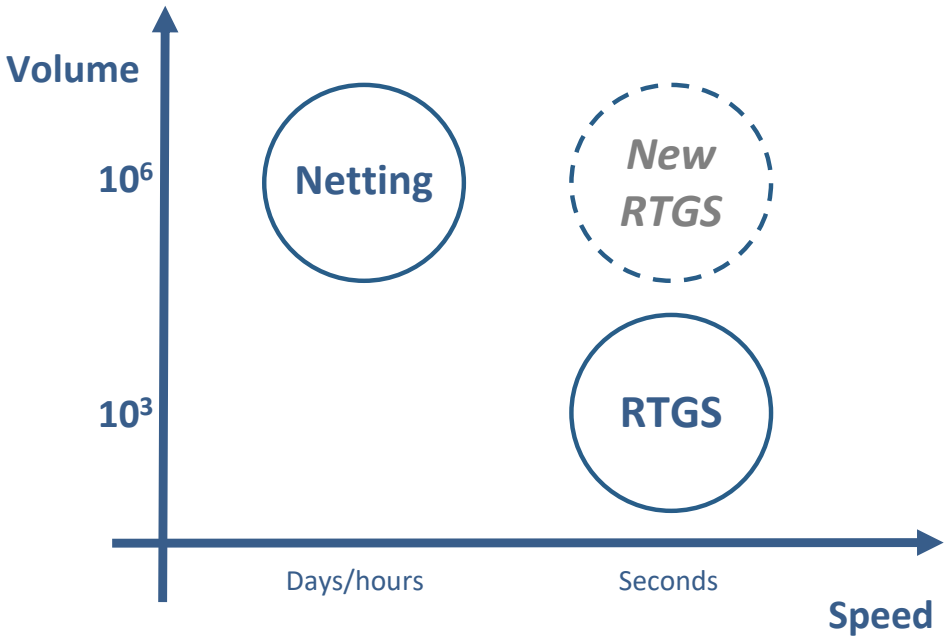

Figure 1. Characteristics of the new RTGS system.

• netting infrastructures can deal with large volumes, but their latency can be hours or even days.

A real-time settlement system for retail payments should be positioned in a third area (see the dotted circle in Figure 1), where low latency ($< 1s$) joins high volumes ($> 10^6$ payments per day): a 'high-volume RTGS' .

1.2 What TIPS is

Target Instant Payment Settlement - TIPS (Renzetti *et al.* 2021; European Central Bank 2018) is a system designed to process retail payments in real time. It works in cooperation with TARGET2, the RTGS system for the euro currency (European Central Bank 2014), which remains the current solution for the settlement of large value payments in real time.

TIPS is the only instant payment system in the EU that can settle payments in central bank money, meaning transactions are immediately settled without requiring any further interaction with the RTGS. TIPS works within the SEPA Instant Credit Transfer (SCT Inst),[3] a scheme launched by the European Payments Council in 2017 to define how commercial banks and the central bank interact to settle instant payments. TIPS was designed and developed by Banca d'Italia which currently operates it on behalf of the Eurosystem.

TIPS goes beyond the limitations of traditional RTGS systems in terms of scalability and availability, allowing banks to settle millions of payments every day round the clock, with just a few seconds' latency. It achieves 24x7 availability and cuts down transaction fees by leveraging a network of Linux/X86 machines. Because of its distributed architecture, it scales out by adding

3. See:https://www.europeanpaymentscouncil.eu/what-we-do/sepa-payment-schemes/sepa-instant-credit-transfer/sepa-instant-credit-transfer-rulebook.

these commodity[4] machines, not by employing more powerful hardware.

1.3 Purpose of this paper

In the following we will discuss how TIPS removed the bottleneck of traditional RTGS systems, by moving from a centralized, transactional engine to an event-driven (Stopford 2018) architecture, which makes no use of database transactions and applies the concept of 'eventual consistency' (Birman and Friedman 1996) instead.

The TIPS design shows how it is possible to break the limitations of a traditional RTGS system by applying the principles and concepts of distributed processing, thereby extending the use of event-driven architectures to the world of payment systems. The result is a new RTGS model, which has the potential to write a new page in the book of payment and market infrastructures.

2 Why not just another RTGS?

The first step in this journey is to understand why traditional RTGS systems, with their centralized database, fall short in scaling up to the volumes requested by the retail payments market, and why there is a need for a completely different solution to support these volumes.

A typical RTGS keeps its data, including account balances, in a centralized database, hosted by an RDBMS,[5] and updated upon every payment using a database transaction. The consistency of account balances is guaranteed at any time by the database, which keeps each account up to date, according to the latest payments that affected its balance. Locks are used to make sure that a balance is not updated concurrently by different transactions, avoiding race conditions[6] (Regehr 2011). So if a balance must be debited as part of a payment, it is also locked to make sure that the availability of the money is not changed after checking it and before the payment transaction is committed.

However, locking prevents things from being done fast. This is particularly true for the retail business, where a relatively small number of accounts (the accounts of the banks)[7] must be updated an extremely high number of times, as each payment that involves two individuals will involve the accounts of their banks at the central bank as well. In this context, locking accounts at database level causes too much contention, and using more threads to run the payment logic in parallel only leads to a lot of deadlocks and timeout issues.

In other words, trying to run concurrent threads to execute multiple transactions at the same time while using locks to prevent race conditions may be fine in many cases, but not for the business of an RTGS, which under a heavy load would strive to update the same account balances (because of a limited number of 'big players') by using different threads, even if parallel processing does not help that much in this case.

2.1 On the way to a distributed system

TIPS uses an innovative approach to consistency, which makes limited use of persistence and no use of transactions. As a distributed system, it trades consistency [8] for availability and is designed with principles of *reactive applications* in mind. So let us first provide an overview of what that

4. Commodity hardware is a computer device that is relatively inexpensive, widely available and designed for general use. It is basically interchangeable with other hardware of its type and thus prevents vendor lock-in.

5. For an overview of relational databases, see: https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

6. A *race condition* occurs when the system's behaviour depends on the sequence or timing of uncontrollable events. It leads to a malfunction when one or more of the possible behaviours is undesirable.

7. As an interbank settlement system, TIPS processes accounts held by banks. On their side, banks are responsible for processing accounts held by their customers.

8. This does not mean that the TIPS architecture abandons data consistency. It simply trades the time it takes to reach full consistency: see after the concept of 'eventually consistent' (Birman and Friedman 1996).

means, before describing the TIPS architecture in detail.

2.2 *ACID* vs *BASE*

There is a 'traditional' approach for designing an operational system. This approach focuses on transaction processing and, as a consequence, on the reliability and consistency of the system. It is mainly based on the well-known ACID properties of database transactions (Roe 2012).

ACID stands for *Atomic*, *Consistent*, *Isolated* and *Durable*. ACID systems rely on relational databases and are heavily based on the concept of *transaction*, which has the following properties:

- atomicity: a transaction is made up of many steps that are executed as a whole. There is no chance for a database transaction to be committed partially: if the transaction is rolled back, each step applied at the time of the rollback operation is undone;
- consistency: data are mostly centralized (or at least 'look' as if they are centralized) and every read receives the most recent write. No *lost update* can occur and no *stale read* is possible. Every transaction leaves the data in a consistent state;
- isolation: concurrent writes on the same data are not possible. Locks are used to prevent race conditions and prevent intermediate changes from being read before they are committed. Transactions can be executed as if they were running alone;
- durability: updates applied by committed transactions cannot be lost even in the event of power loss or other system failures (i.e. the state is persistent).

So what does it mean to build a system that can no longer rely on the above guarantees? Without the 'safety net' of a relational database, such a system needs to address the following:

- when atomicity is lost and some error occurs that does not make it possible to complete a series of steps, compensating actions must be implemented to undo the ones that have been already completed. The system must be prepared to handle failures in compensating actions as well, resuming those that failed later or, better, reapplying the whole chain of actions from the beginning if all of them are idempotent (i.e. if they can be applied again multiple times with no side effect);
- when data are no longer centralized, but rather are distributed and/or replicated across multiple machines, the system must be able to work even in event that a subset of the data is not available (e.g. because of the temporary unavailability of a part of the system) and/or must be able to cope with multiple copies of the same data, some of which may not be up to date. When strong consistency guarantees are required by the application domain, the system must make sure that every read gets the most recent write, or stop working (Kingsbury 2018), (Dziuma *et al.* 2011);
- the way that isolation is guaranteed, with no centralized database, changes together with the architecture itself, which moves from a *shared-state* model – wherein concurrent threads update the same data and use database locks to isolate each other – to a *share-nothing* (Stopford 2009), *message-passing* model – wherein isolated actors run on separate machines, and each one updates a local store based on messages exchanged with the others.

With a nod to chemistry, the state of the art of this matter sometimes uses the acronym BASE (as opposed to ACID) which stands for *Basically Available, Soft-state, Eventually consistent*, and refers to the following:

- basic availability: the data may be partly unavailable, or the available copy may not be up to date (a read may not receive the result of the most recent write);
- soft state: the state of the system, being the collection of multiple state stores held locally by several actors, could change over time and even without further input, as a consequence of asynchronous message passing between these actors;
- eventual consistency (Birman and Friedman 1996): multiple copies of the same data, replicated across multiple nodes, sooner or later evolve to become the same (in the absence of further

input). Similarly, different data held locally by separate actors eventually evolve to be consistent with one another.

The main concept behind the BASE acronym is that in a distributed system with no centralized state, there is a tradeoff between consistency and availability, which is clearly described by a theorem of distributed computing.

2.3 The CAP Theorem

The CAP theorem (Gilbert and Lynch 2002), (Nazrul 2018), (Whittaker 2014), (Hale 2010), (Brewer 2012) is usually described as follows: when building a distributed system, one can only pick any two of the (desirable) properties shown in Figure 2, but not all of them.

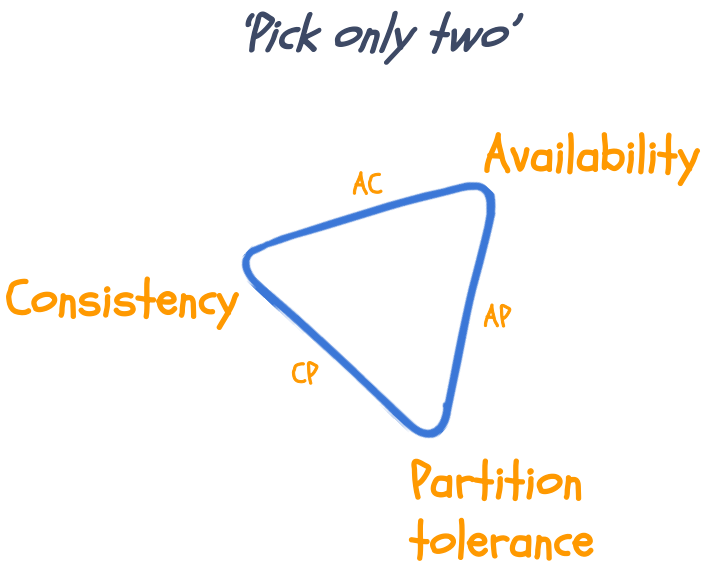

Figure 2. CAP theorem - *A distributed system can only preserve two properties from: Consistency, Availability and tolerance to network Partitioning.*

Availability is achieved by replicating data across multiple nodes, while consistency is achieved by updating all nodes before allowing reads on the updated data, or equivalently by making sure that reads can only get the data out of a node that has received all the updates.

In the event of a network failure which prevents communication between different nodes, there is no chance of keeping the system consistent and available, because data replication is impossible. So one may choose consistency over availability and thus stop the service (CP mode, see Figure 2), or vice-versa availability over consistency, and have diverging copies of the same data which need to be reconciled later (AP mode). The only way to achieve both availability and consistency is to avoid any chance of network partition, that is, to use a traditional centralized system (AC mode).

As the consistency of account data is of the utmost importance for an RTGS, a distributed RTGS (such as TIPS) must prefer consistency and stop the service as soon as the outage of network resources no longer allows its accounts to be kept consistent.

2.4 The Reactive Manifesto

Some of the concepts described above are well applied by the *reactive applications*: these are listed in Figure 3 and were defined a few years ago in a *Reactive Manifesto* (Bonér *et al.* 2014) (Bonér 2014),

undersigned by a variety of architects known in the software industry for promoting patterns and frameworks for distributed programming.

Figure 3. Reactive manifesto - Principles and concepts of reactive systems.

In contrast with traditional centralized systems, reactive applications use loosely-coupled actors that cooperate together within a message-driven model, sharing no centralized mutable state and leveraging redundant processing to maximize responsiveness and resilience.

The principles and concepts of reactive programming confirm the need for a different approach when applications have to cope with high-throughput and low-latency requirements, while being available nearly 100 per cent of the time. To some extent, these principles and concepts have also been applied to the design of an RTGS such as TIPS.[9]

3 The new settlement engine architecture

The proposed architecture for a settlement engine that needs to handle a huge volume of payments (which is unmanageable for any traditional RTGS design) uses parallel logic where this brings some advantages and serial processing where it does not. This means it handles in parallel only the part of processing that does not require synchronization on any shared resource, as this usually acts as a bottleneck for parallel processing.

Moreover, persistence (and data replication, as a valuable alternative used by distributed systems to keep the information 'safe' even when it is not stored on disk) is mixed with usage of in-memory processing. This mix must be carefully weighted to prevent any kind of inconsistency.

In a traditional system, persistence and database transactions enable the identification of the points in time when the system reaches a consistent state, which means that all the data collected and stored are consistent with all the input received and the output sent through the system interfaces. When leaving ACID transactions to adopt the BASE model, we are essentially moving from a *consistent* system to an *eventually consistent* (Birman and Friedman 1996) one. In this architecture, this occurs through the creation of an ordered sequence of requests, which is stored 'persistently' (and/or replicated on multiple machines, to make sure it is never lost) and

9. Reactive systems are known to be *responsive*, *resilient*, *elastic* and *message-driven*. The design we propose here covers all these definitions except for *elasticity*, as the current implementation of TIPS is not able to trigger the automatic provisioning of new servers to cope with varying workloads. However, this feature could be added on top of the proposed design without changing its current shape.

distributed to multiple 'Settlement Cores'. By reading and processing all payment requests *in the same order*,[10] each Settlement Core *eventually* reaches the same state, which is a meaningful embodiment of *eventual consistency*.

The sequential ordering of the input stream gives assurance that any information that makes up the state of the system can be rebuilt by *replaying* the stream, and therefore can also be *safely lost*. In other words, any information that makes up the state of the system does not need to be persistent, while the input stream does.

To make the replay technique meaningful, every Settlement Core must be totally isolated, and its internal state must only be affected by the input stream: only this property ensures that, given any two sequence numbers x and y (with $x < y$), if the system is in state s_x and it replays the input stream from element number x up to element number y, then it reaches the state s_y.

3.1 Parallel vs Sequential processing

When processing multiple workloads, concurrent threads make it possible to scale out,[11] but are really only effective when these threads experience poor or no contention on shared resources. In TIPS, this is true when doing the following:

Before settlement:

- validating input, executing checks that can be verified against each message alone, without modifying any mutable state;
- running uniqueness checks, reading a shared repository of payment requests by key, to make sure they are not already present, before they can be accepted;
- keeping track of accepted payments in the same repository, with different threads writing different records (this is done together with the uniqueness check in one step);
- sending payment requests to the subsystem that is responsible for creating the *journal*, i.e. the ordered sequence of payment requests to be processed by the Settlement Cores.

After settlement:

- updating the shared repository of payment requests with settlement results, accessing payment information by key;
- informing payment banks about these results by sending out response messages.

10. More precisely, TIPS settlement implements the following *deterministic finite state machine*, a system:

$$M = \{I, O, S, f, g\}$$

where:
$I = \{i_1, i_2, ..., i_n\}$ is the finite set of all the possible input symbols,
$O = \{o_1, o_2, ..., o_n\}$ is the finite set of all the possible output symbols,
$S = \{s_1, s_2, ..., s_n\}$ is the finite set of all the possible states,
$f : I \times S \rightarrow O$ is the function linking input values to output values,
$g : I \times S \rightarrow S$ is the transition function of system internal states.

Given an ordering, strict and total, defined on I and S, then for any element $\lambda \in \mathbb{N}\ (\lambda < n)$:

$$O(\lambda) = f(I(\lambda), S(\lambda)) \text{ and}$$
$$S(\lambda + 1) = g(I(\lambda), S(\lambda)).$$

So, both the internal status of M and its output depend solely on the input and previous status and they do not depend on any other condition, such as time.

11. To *scale out*, or to *scale horizontally*, means to bear an increasing workload by just adding more nodes, not by employing more powerful hardware. To *scale up*, or to *scale vertically*, means the opposite. Only being able to scale up is, in principle, a flaw: increasing the computational resources assigned to the existing node could encounter technological limits, as the well known Moore's law predicts.

The above tasks are under the responsibility of 'Message Router' components. Multiple Message Routers can be installed on different machines, and each of them can be configured to run multiple *workers*. These workers handle the tasks above in parallel, with no need to synchronize with each other.[12] The only shared information is the reference data[13] required to perform validation checks, but there is no need for synchronization and no use of locking, as it is only available for read. Moreover, writes made by different workers to keep track of input payments are independent, and do not block each other. This means again there is no bottleneck, no race condition, and no deadlock.

The duty of Settlement Cores is quite different, as they need to update a limited number of bank accounts as a consequence of handling quite an extended volume of payment requests. This is specific to the nature of the service implemented by an RTGS: as every input payment is a transfer of money between two bank accounts (and banks are not that many), if a huge volume of requests is received in a short time frame, an RTGS needs to update the same balances *again and again*. Of course, account balances are a *shared mutable state*, which means that locking is needed if this state is updated concurrently by multiple threads. As a result, increasing volumes in any traditional RTGS system, which holds account balances in a centralized RDBMS, sooner or later means facing deadlock and timeout issues.

For this reason, the solution applied by Settlement Cores, which are responsible for updating account balances, is based on *serial processing*: every core is a single thread, which can only process one payment at a time, so that no synchronization or locking is needed. Moreover, the processing is extremely fast as it works in memory.[14]

Figure 4 shows parallel Message Routers (three instances are shown in the figure, but this just an example: the number can vary depending on the needs) and the sequential Settlement Core.

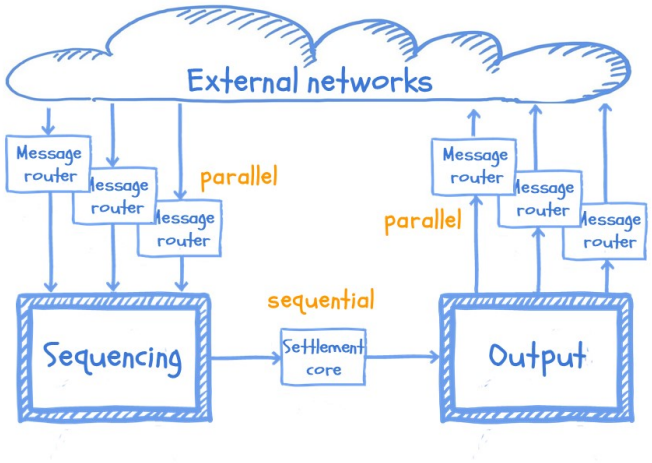

Figure 4. Parallel and Sequential streaming - The system creates the journal (including payments, orders and anything that can change the internal status of accounts and balances) as a sequence (serialization).

12. As we said in note 10, there is no need for each worker to know anything else except its input ($I(\lambda)$) and internal status ($S(\lambda)$). Consequently, there is no need to share information among parallel instances.
13. These is static information collected externally and periodically injected into the system.
14. This makes it possible to *scale up* by exploiting processor capacity.

3.2 Disk vs Memory

To make it possible for Settlement Cores to hold updated account balances in memory, thereby dramatically increasing the speed of settlement and leveraging a single-threaded engine, payment requests are sorted during input phase, and each of them is assigned a *unique sequential number*.

The ordered sequence of input requests is stored in a *journal* that is read by Settlement Cores to get the payment requests. As long as this journal is stored in persistent storage, the status of account balances held by Settlement Cores can be lost, as the journal keeps track of anything that is needed to build it again. Consequently, the use of in-memory processing at the Settlement Core level becomes more than meaningful.

Message Routers are responsible for writing the journal right after any other task is completed (see above). The point in time when a router instance writes a record in the journal draws an imaginary line between parallel processing, which happens before, and serial execution, which is done later.

Writing a journal, as a sequence of events that is shared by multiple peers, is something that cannot scale out, of course. Every writer instance needs to get exclusive access to some write handle and append a record at the end of the sequence, something that cannot be done in parallel. So this is actually the first point of the architecture where serial processing is applied.

Moreover, the fact that sequential disk access and random memory access have a comparable performance has been known by the scientific community for a long time.[15] This means that a single writer can serve incoming write requests from multiple Message Routers and compose the journal using a single-threaded component while being almost as fast as an in-memory settlement engine that reads the journal. Intuitively, the speed of this single-threaded component easily sets an upper bound for the throughput of the overall system, but this does not threaten the effectiveness of the solution in use for Settlement Cores, as both the latter and the component that writes the journal can be extremely fast.[16] More interestingly, the upper bound set by this component is significantly above that of a traditional RTGS.[17]

3.3 From a Funnel to a Pipe

In summary, if an old-shaped RTGS acts like a funnel, striving to synchronize access to shared account balances through database locking, the new engine can be seen as a pipe, into which the instructions move one at a time, coming out again very quickly. The speed is given by the absence of contention and by the in-memory processing, with no on-disk persistence. To balance the volatility of its state, the settlement engine has the chance to rebuild it at any time, by re-processing the journal. As described later on, there are ways to speed up this phase, as the replay of the whole journal after a failure would otherwise be impractical.

As this state must depend only on the incoming sequence of events stored in the journal, this must also include, in addition to payments, any other 'instruction' that changes the settlement behaviour. To provide an example of such a rule, the journal must also include instructions that change the status of accounts or balances, such as the blocking or unblocking of accounts, the change of credit limits and so on. Any update of the information that potentially affects the settlement can only be carried out through the journal processing, so that it can be replayed.

Similarly, queries that read status information from Settlement Cores are written to the journal: this way, the response can be consistent with an exact point of processing, which is the one between

15. See https://queue.acm.org/detail.cfm?id=1563874.

16. As described later on, in TIPS this component is based on a cluster of message brokers implemented by a dedicated middleware. At any point in time, the journal is written only by one of these brokers, but it is also replicated to the other ones, so that in the event of a failure (Schneider 1990) another broker can replace the failed one and start writing.

17. Tests have shown that this approach can scale up to 1 billion payments/day.

the preceding payment and the following one. In such a design, the sequence number used in the journal is the only meaningful time reference, as it is an event y occurring later than event x if and only if its sequence number is higher ($\lambda_y > \lambda_x$).

4 Continuous availability and resilience

A service that must be available on a 7/24/365 basis must be able to cope with failures in any of the several layers underneath its covers, from bare metal to middleware components. As a distributed system, TIPS takes advantage of multiple nodes to ensure the continuous availability[18] of the service in the event of failures. This is done in different ways, based on the role that is played by different nodes.

4.1 Message Router Continuity

In the event of a failure of a Message Router node, the remaining nodes can take over transparently, just taking care of the additional workload, which is evenly distributed across them.

Any input message that was in flight at the moment of the failure may be lost. However, this cannot lead the system to an inconsistent state: messages not yet processed can be submitted again to the system. Output messages, such as notification of settlement results, may be lost as well;[19] nevertheless all messages are safely stored and, in the event of transmission failure, the external actor (bank) is given a specific function to retrieve ('pull') any missing message.

4.2 Settlement Core Continuity

As discussed previously, Settlement Cores can recover from a failure by replaying the journal. However, as the replay of the journal from the beginning takes too long just after a few days since the system was started, local snapshots of memory structures are triggered regularly, keeping the need for replaying the journal to a minimum.[20] So, the tasks needed to start a new instance of the core, ensuring the integrity of its status, are the following:

- load the latest snapshot into memory, reading the sequence number of the last instruction processed before taking the snapshot (s_{snap});
- replay the journal starting from sequence s_{snap} and until the last available entry.

The duration of the procedure above depends on the input rate, on the frequency of the snapshots and on the speed of the replay logic. Until this procedure has been completed, a core cannot process new payment requests.

This means that, to avoid any service disruption, it is highly desirable never to run this procedure, even in the event of a core failure. To achieve this, multiple cores (see Figure 5) can be deployed to concurrently read the same journal and process the same payment requests. Unavoidably, each one of them will run at a different speed but, sooner or later, all of them will reach the same status, which means they will be *eventually consistent* (Birman and Friedman 1996).

The cooperation of Settlement Cores is rather different from what happens on Message Routers, as each core processes the entire workload, not just a subset of it. In principle, one core may be considered *active* and produces 'real' output (i.e. the responses that are promoted to the output stage), while all other instances just keep up with the updates received from the journal, being ready to replace the active instance in the event of failures. We may call the latter *hot-ready* instances.

18. TIPS is a *crash fault tolerant* (CFT) system. More precisely, given the taxonomy provided by (Barborak, Malek and Dahbura 1993), TIPS complies with the *Omission Fault* class and partly complies with *Incorrect Computation Fault* class.

19. The external networks consistently support this *at-most-once* message delivery approach: there are no guarantees that any outgoing message will be delivered to the final recipient.

20. The speed of snapshots is critical to avoid affecting the latency of incoming payments, as Settlement Cores cannot change the state while it is being serialized on local disks.

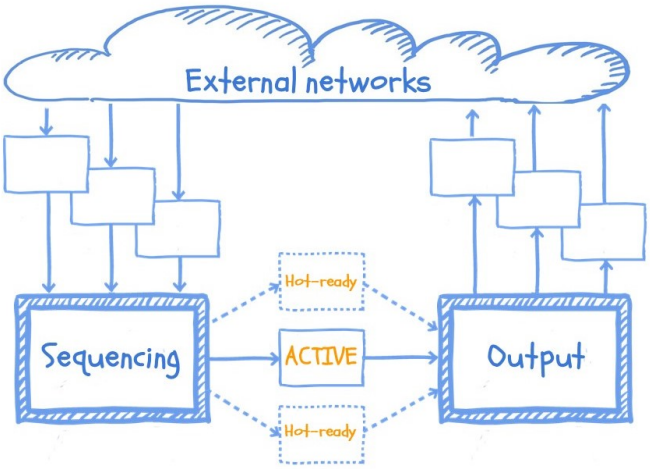

Figure 5. Parallel and redundant processing - After incoming messages have been validated and normalized, the system replicates the very same sequence to several settlement nodes. Only one of these nodes is 'active' (i.e. it is producing valid output), while the others are only working to be ready to take over the active node if it crashes.

It is worth noticing that what has been described is slightly different from a typical *master-slave* architecture, in which one or more *slave* nodes keep up with the updates coming from a *master* node responsible for the production and distribution of results. In this case, all the nodes are equivalent, because they all build their state independently by reading all they need from a third-party, which is the journal. Thanks to this, it is possible to take a further step.

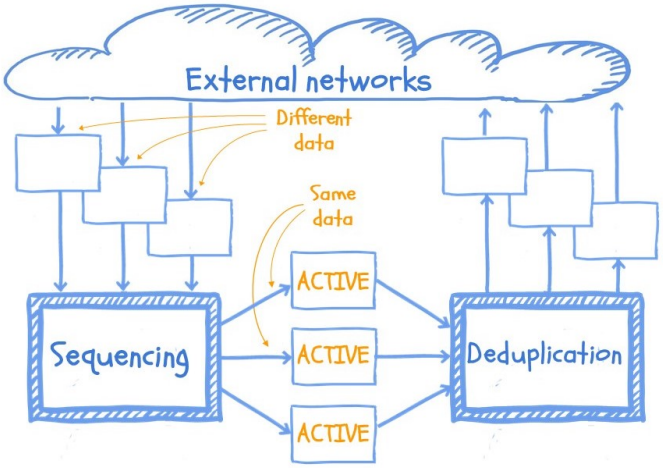

Figure 6. *Masterless* architecture - All the sequential components play an 'active' role and each of them takes on the whole system workload. On the right, a deduplication stage sends out only the first result received for any given sequence number.

By replacing the output stage with a deduplication stage, it is possible to avoid having to elect a new master whenever the active node fails (or is just shut down for maintenance). Because the results produced by multiple cores are equal, duplicates may simply be removed before these results are promoted to output, which is exactly what is done by the deduplication stage. This stage relies on the strict ordering of settlement results, which is ensured by the ordering of the journal and by the fact that Settlement Cores are single-threaded. Because of the above, if the result of sequence s_y has been received at this stage, then the result of any sequence s_x can be dropped if

$s_y > s_x$.[21] Figure 6 shows the system with all the Settlement Cores playing an 'active' role.

The main advantage of using such a *Masterless* approach is simplicity, as implementing a *quorum-based* consensus[22] (Pease, Shostak and Lamport 1980) is not easy. Rather than relying on existing clustering frameworks (e.g. Hazelcast, Zookeeper), TIPS keeps this implementation out of the boundaries of the application, leaving it to the middleware components it uses, for different reasons, at different stages of the architecture.

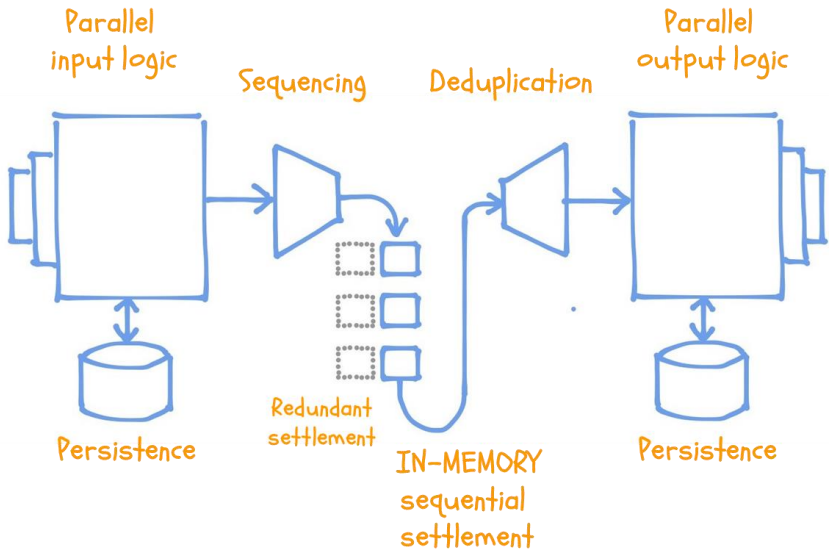

Figure 7. Overview of the settlement architecture.

Figure 7 shows an overview of the settlement architecture. From the left:

- components responsible for running parallel tasks validate incoming messages. They store data ion persistent storage (they read data as well, to support duplicate checks);
- then the flow gets (strictly) sequential and the processing is handed over to sequential components, which process payments one at a time (but very fast) and keep track of account balances in memory; at this stage, redundant nodes perform the same tasks multiple times;
- finally, duplicate results are dropped and responses are sent to the external recipients. This process, which is again done in parallel, also updates payment data on persistent storage, to keep track of all the instructions that have changed the system state (for investigation queries, reports and so on).

5 Middleware components

During the first phase of the design of the system, several concerns came up about the consistency of distributed data, and the way consensus should have been reached in the event of node failure. Later, the design of the application changed in the way just described, delegating these duties to dedicated middleware, which runs over a distributed cluster of several nodes. In the following, the most important middleware components used by TIPS are briefly described regarding their

21. This is also implemented by using a cluster of message brokers, but this time the entire set of settlement results is partitioned in such a way that multiple results produced for the same payment are written into the same partition and can be managed by the same broker and delivered to the same consumer.

22. In distributed systems, to achieve overall system reliability in the event of some process fails, coordinating processes reach a consensus on their states, or agree on some data values.

function and role.

5.1 Message brokers

Message Routers and Settlement Cores exchange messages through open-source *message brokers*, running over a cluster of nodes. By cooperating together, message brokers maintain a *distributed commit log* (Kreps 2016), which is the means for the implementation of the TIPS journal.

Brokers support the publish-subscribe model and allow delivered messages to be *replayed* upon request of subscribers. By relying on this feature, actors that read messages can resume the processing and rebuild their state after a failure.

Messages are appended to *partitions*[23] by producers and cannot be modified, but only read by subscribed consumers in the same order. At any point in time, every consumer has read a specific partition up to some point, which is known and managed by the consumer itself. After a failure, the consumer can resume the processing by looking for messages produced after the last known position, thereby recovering lost information.

A *single-partition* thread of input messages (those sent by Message Routers to Settlement Cores) implements the journal.[24] This partition is written by Message Routers through requests to the only broker that is on duty to write it, replicated to all brokers in the cluster, and read by all Settlement Cores. Replication allows brokers to provide fault tolerance capabilities, as they can take over from each other in the event of failures, because they all own a full copy of the input partition.

According to how message brokers are configured, the broker that is on duty for writing the input partition commits a write when it was successfully replicated to the majority of the nodes, and Settlement Cores can only read *committed* writes.

Settlement results are written to multiple output partitions. For a payment request, duplicate results produced by multiple Settlement Cores are written to the same output partition, so that they can be deduplicated by the same consumer, as messages of the same partition are read by the same consumer. Duplicates are detected according to the sequence number in the journal, and thanks to the strict ordering of the input partition, which is preserved by Settlement Cores, the deduplication logic does not need to 'remember' all received messages, but only the sequence number of the last payment request for each output partition.

This makes it possible to keep Settlement Cores equal, without requiring a *master* to be elected among them. Masters do exist, but only in the message broker middleware.

5.2 Persistence cluster

So far, we have extensively discussed the chance to keep account balances only in memory, which is made possible by a journal. However, the same option is not viable for payment results, because even though these results could be conceptually held in memory for the same reasons, they must be offloaded somewhere to be read or browsed even hours or days after the settlement, and of course, keeping all the payments settled in hours or days in memory is impractical.

Storing payments in a database does not raise the same contention issues for which an RDBMS was deemed incapable of supporting the volume of retail markets (which was the driver of the choices described above). However, we wanted to implement a database that could scale-out linearly by just adding more hardware, also getting the benefit of improved resilience, so we opted for a distributed *NoSQL*[25] *database* (Strauch 2018) to implement the payment repository, making

23. The message stream is divided into partitions to enable process parallelization: each partition is controlled by a separate machine to allow multiple consumers to read them in parallel.

24. The single partition preserves the ordering of the journal. Please note that even in this case, there is no *single point of failure* as the partition is handled by the cluster of brokers and, as such, it is replicated across the cluster.

25. The term refers to a new type of database - 'nonSQL' or 'non relational' - that goes beyond the traditional relational databases (i.e. based on tabular relations), and whose importance has increased with the emergence of big data and web

sure that the work carried out by Message Routers in collecting results could scale up through parallel processing.

The distributed database used for this task is a NoSQL «speed at scale» key/value database that implements data *sharding* (Zhang *et al.* 2015), i.e. distribution of data across multiple servers, with a customized level of replication. Records are organized in partitions and partitions are distributed to servers, so that at any point in time there is only one master for any partition, but different partitions are managed by different masters (no single point of failure, no bottleneck to break the scalability of the cluster).

Finally, with reference to the CAP Theorem, TIPS elects consistency in the event of network splitting,[26] acting as a CP (*consistent* and *partition tolerant*) type of system (see Figure 2). The distributed database is configured to supports this choice: when a network split occurs, it stops working and chooses to make the service unavailable, to preserve its consistency. This is essential, in order to make sure that reads are *linearizable*,[27] and to prevent any chance of ending up with two different copies of the same data in the event of a network split.

6 Performance

Before launching TIPS, a dedicated test campaign was carried out, aimed at assessing the ability of the system to meet, among others things, its volume requirements. The campaign did not reveal any overall limits, except for those deriving from the production environment as regards its current setup and sizing, i.e. according to the specific hardware and software configuration that was chosen to bear the expected workload in production.

The system configuration (see Figure 8) used for the volume test campaign can be described as follows:

Message injection
- 4 input feeders (simulating payment originators)
- 4 responders (simulating payment beneficiaries)

Application layer
- 4 *Message Router* nodes (4 virtual CPUs, 64 GB RAM)
- 4 *Settlement Core* nodes (4 virtual CPUs, 256 GB RAM)

Middleware layer
- 6 NoSQL servers (12 virtual CPUs, 256 GB RAM)
- 6 message brokers (8 virtual CPUs, 64 GB RAM)
- 4 queue managers (8 virtual CPUs, 32 GB RAM)

6.1 Capacity at middleware level.

The settlement throughput is measured taking into account that, for each instant payment, there are two settlement phases, the first required to reserve funds and the second to finalize the payment after being confirmed on the beneficiary side. For each instant payment, four write operations and one read operation are performed on the NoSQL database. Moreover, two input messages and eight output messages must be written by the messaging system.

According to the average and peak volumes to be supported by TIPS, which are 500 payments/s and 2000 payments/s respectively, the middleware components must be able to support the throughput in Table 1.

applications.

26. A network split is a failure that generates a decomposition of the network into independent subnets, with every node belonging to a subnet only being able to communicate with the nodes in the same subnet.

27. See https://jepsen.io/consistency.

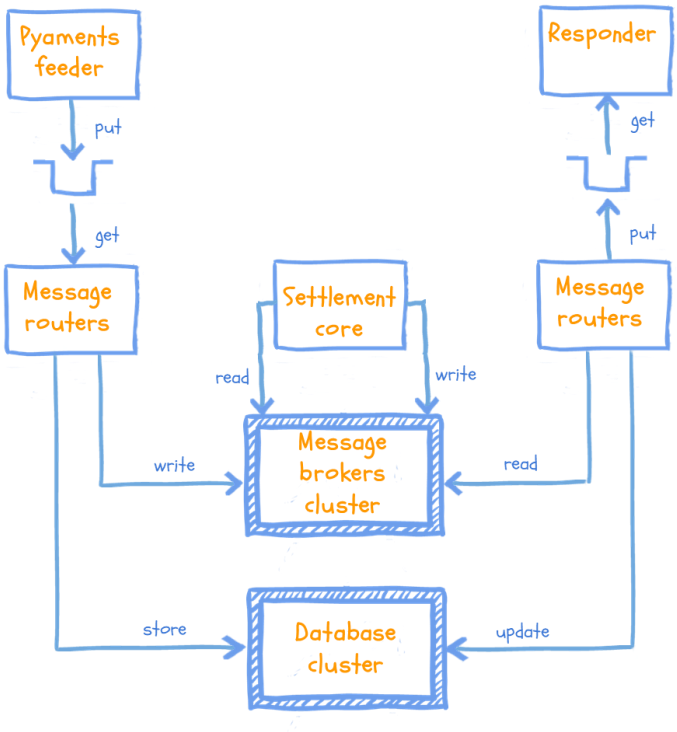

Figure 8. System configuration for performance tests.

Table 1. Throughput that TIPS must support (n_r/s database reads per second, n_w/s database writes per second, n_i/s number of input messages per second, n_o/s number of output messages per second).

Scenario	n_r/s	n_w/s	n_i/s	n_o/s
Average workload	500	2000	1000	4000
Peak workload	2000	8000	4000	16000

Tests have shown that, in the chosen configuration, the message brokers cluster was able to deliver up to 100000 messages/s per single partition, while the database cluster could perform up to 120000 writes/s and 30000 reads/s.

In both cases, performance was well beyond the goals described above. However, the throughput measured for message brokers needs a special mention, as it limits the throughput of the overall system. As input messages are written to a single partition, which is handled by one only broker at a time, TIPS cannot process payments faster than the maximum input message rate.

More in detail, with two input messages for every payment, a throughput of 100000 messages/s on a single partition means that with the current production setup, TIPS speed is limited to 50000 payments/s. This sets an upper bound on the TIPS throughput.

6.2 Capacity at application level.

Concerning the scalability of the whole system, Figure 9 shows the results of the measurements taken during the test campaign and the corresponding overall throughput, including both middleware and application performances.

The first part of the curve (i.e. below 7000 payments/s), denotes a good similarity between the

TIPS architecture and a generic $O(n)$ algorithm or linear-time[28] (Du and Ko 2000) algorithm within the given area of application.

The bending point shown between 7000 and 8000 payments/s is where the system saturates, meaning it can no longer cope with the message input rate. This point is determined by the actual configuration of the system, not by the architecture itself, as it depends on the number of horizontally-scalable nodes (those that host Message Router instances). In fact, the throughput of each Message Router, on this hardware, is slightly below 2000 payments/sec. By adding more Message Router nodes, the saturation point shifts to the right.

On the other hand, Settlement Core nodes have been found to be working well below their limits in the current setup. Because the processing is performed in memory, the actual speed of a Settlement Core goes beyond the limits of the message broker that feeds messages into it, which is the only bottleneck of the system as already described. The upper limit of a Settlement Core cannot be found by using a full system configuration like the one just described, because the limit of the messaging system is found earlier, but tests with software stubs have shown that in-memory processing can easily scale up to 500000 payments/s and more.

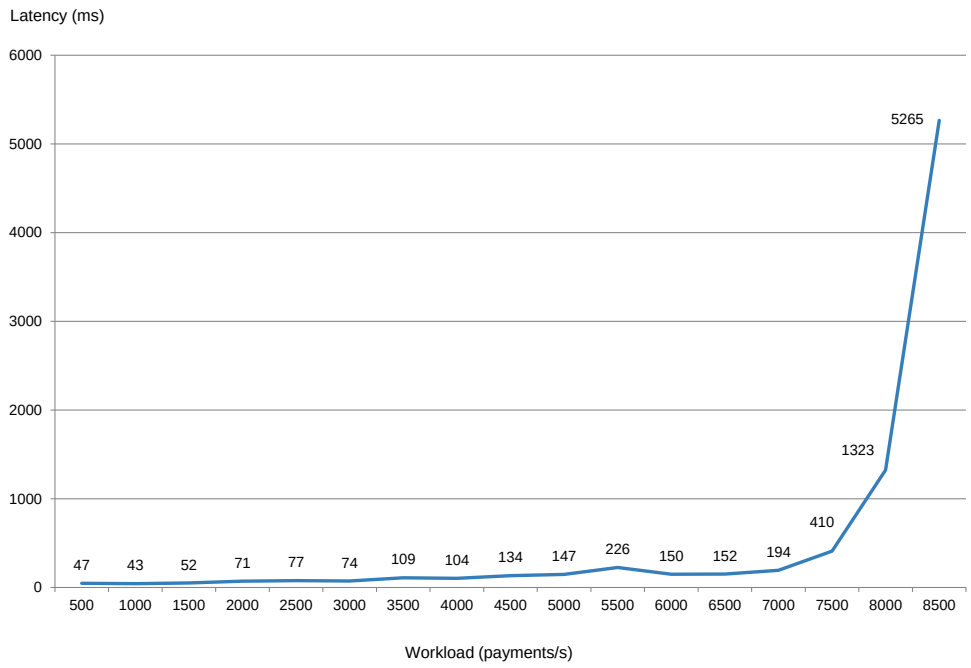

Figure 9. Scalability curve - latency by workload.

7 Conclusion

When designing TIPS, our focus was on removing the typical limitations of a traditional RTGS system in terms of scalability and availability.

The requirements described the need for a system design able to support the settlement of millions of payments per day, with a few seconds latency, while still keeping costs as low as

28. An algorithm is said to take linear time, or $O(n)$ time, if the running time increases at most linearly with the size of the input. This is the best possible time complexity when the algorithm has to read its entire input sequentially.

possible. This led us to the idea of designing a distributed system that, by using a network of Linux machines, was more appropriate than using a centralized system, which is the typical choice for the implementation of an RTGS.

We looked for a state-of-the-art architecture to implement a system with the appropriate level of consistency on a distributed cluster, in order to leverage the power of a network of machines without losing the reliability of the data, which was not questionable. It was a new challenge, a real breakthrough for the development of systems of this kind, but also a disruptive shift for the operational model, due to the need to keep the service up and running around the clock.

At the end of this journey, we completely abandoned the centralized model, and faced the difficulties (Currie 2017) of distributed systems (Lamport 1978), which was the only way to give an answer to all of our questions.

We eventually reached a system design that does not make use of parallel processing when it cannot provide its usual benefits because of resource contention, and rather exploits the power of processors through sequential computing, which combined with in-memory data structures, achieves outstanding performance of the settlement logic without sacrificing reliability.

The resulting architecture suggests a completely new model, which replaces centralized systems with a network of highly redundant, inexpensive, interchangeable computers and this could be taken as a reference for other RTGS[29] systems with challenging requirements in terms of scalability and availability and for other mission-critical systems in the world of financial institutions.

29. A typical RTGS for large value provides functions for liquidity optimizations, such as queues for payments with temporary lack of funds or algorithms for preventing or solving gridlock conditions. TIPS, as a retail service, does not have such facilities but, nevertheless, its architecture does not prevent from 'plugging' those new components on top of its settlement engine.

References

Barborak, M., M. Malek and A. Dahbura. 1993. “The Consensus Problem in Fault-Tolerant Computing”. *ACM Computing Surveys* 25:182–184. https://www.researchgate.net/publication/220566027_The_Consensus_Problem_in_Fault-Tolerant_Computing.

Birman, K., and R. Friedman. 1996. “Trading Consistency for Availability in Distributed Systems”. *Cornell University, Ithaca, NY,* https://ecommons.cornell.edu/handle/1813/7235.

Bonér, J. 2014. “Reactive Manifesto 2.0”. Accessed: 2021-01-05, https://www.lightbend.com/blog/reactive-manifesto-20/.

Bonér, J., D. Farley, R. Kuhn and M. Thompson. 2014. “The Reactive Manifesto”. Accessed: 2021-01-05, https://www.reactivemanifesto.org/.

Brewer, E. 2012. “CAP Twelve Years Later: How the "Rules" Have Changed”. *Computer, IEEE Computer Society* 45 (02): 23–29. https://doi.org/10.1109/MC.2012.37.

Chakravorti, S. 2000. “Analysis of systemic risk in multilateral net settlement systems”. *Journal of International Financial Markets Institutions and Money* 10 (1): 11–12. https://www.sciencedirect.com/science/article/pii/S1042443199000220.

Committee on Payment and Settlement Systems - Technical Committee of the International Organization of Securities Commissions. 2012. “Principles for financial market infrastructures”, 149. http://www.bis.org/cpmi/publ/d101a.pdf.

Committee on Payment and Settlement Systems of the central banks of the Group of Ten countries. 1997. “Real-Time Gross Settlement Systems”. *Bank of International Settlements, Basle,* 10. http://www.bis.org/publ/cpss22.pdf.

Currie, A. 2017. “Distributed Systems Are Hard”. Accessed: 2021-01-05, https://thenewstack.io/distributed-systems-hard/.

Du, D.-Z., and K.-I. Ko. 2000. “Theory of Computational Complexity”. *John Wiley Sons, Hoboken, New Jersey.*

Dziuma, D., P. Fatourou, E. Kanellou and A. Kasra. 2011. “Survey on consistency conditions”. Accessed: 2021-01-05, https://projects.ics.forth.gr/tech-reports/2013/2013.TR439_Survey_on_Consistency_Conditions.pdf.

European Central Bank. 2014. “What is TARGET2?” Accessed: 2021-01-05, https://www.ecb.europa.eu/paym/target/target2/html/index.en.html.

———. 2018. “The Target Instant Payment Settlement system”. Accessed: 2021-01-05, https://www.ecb.europa.eu/paym/target/tips/html/index.en.html.

Gilbert, S., and N. Lynch. 2002. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. *ACM SIGACT News (Association for Computer Machinery - Special Interest Group on Algorithms Computation Theory), New York, NY* 2 (2002-2): 51–59. https://dl.acm.org/citation.cfm?id=564601.

Hale, C. 2010. “You Can’t Sacrifice Partition Tolerance”. Accessed: 2021-01-05, https://codahale.com/you-cant-sacrifice-partition-tolerance/.

Kingsbury, K. 2018. “Strong consistency models”. Accessed: 2021-01-05, https://aphyr.com/posts/313-strong-consistency-models.

Kreps, J. 2016. “The Log: What every software engineer should know about real-time data’s unifying abstraction”. Accessed: 2021-01-05, https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying/.

Lamport, L. 1978. “Time, clocks, and the ordering of events in a distributed system”. *Communication of the ACM, New York, NY* 21 (7).

Nazrul, S. S. 2018. “CAP Theorem and Distributed Database Management Systems”. Accessed: 2021-01-05, *Towards Data Science,* https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e/.

Pease, M., R. E. Shostak and L. Lamport. 1980. “Reaching Agreement in the Presence of Faults”. *Journal of the ACM (JACM), New York, NY* 27 (2): 228–234.

Regehr, J. 2011. “Race Condition vs. Data Race”. Accessed: 2021-01-05, *Embedded in Academia,* https://blog.regehr.org/archives/490.

Renzetti, M., S. Bernardini, S. Marino, L. Mibelli, L. Ricciardi and G. M. Sabelli. 2021. “TIPS - TARGET Instant Payment Settlement The Pan-European Infrastructure for the Settlement of Instant Payments”. *Mercati, infrastrutture, sistemi di pagamento (Markets, Infrastructures, Payment Systems), Questioni istituzionali (Institutional Issues), Banca d’Italia, Roma,* no. 1, https://www.bancaditalia.it/pubblicazioni/mercati-infrastrutture-e-sistemi-di-pagamento/questioni-istituzionali/2021-001/en-MIS-20210129.pdf?language_id=1.

Roe, C. 2012. “ACID vs. BASE: The Shifting pH of Database Transaction Processing”. Accessed: 2021-01-05, https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/.

Schneider, F. 1990. “Implementing fault-tolerant services using the state machine approach: a tutorial”. *ACM Computing Surveys* 22 (4). https://www.cs.cornell.edu/fbs/publications/smsurvey.pdf.

Stopford, B. 2009. “Shared Nothing v.s. Shared Disk Architectures: An Independent View”. Accessed: 2021-01-05, http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/.

———. 2018. “Designing Event-Driven Systems”. *O’Really media, Sebastopol, CA,* https://www.oreilly.com/library/view/designing-event-driven-systems/9781492038252/.

Strauch, C. 2018. “NoSQL Databases (white paper)”. Accessed: 2021-01-05, http://www.christof-strauch.de/nosqldbs.pdf.

Whittaker, M. 2014. “An Illustrated Proof of the CAP Theorem”. Accessed: 2021-01-05, https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/.

Zhang, H., G. Chen, B. Chin Ooi, K.-L. Tan and M. Zhang. 2015. “In-Memory Big Data Management and Processing: A Survey”. *IEEE Transactions on Knowledge and Data Engineering* 27 (7). https://ieeexplore.ieee.org/abstract/document/7097722.