Trantor: Modular State Machine Replication

Matej Pavlovic

September 4, 2023

Abstract

We present Trantor, a modular state machine replication (SMR) system. It decomposes the concept of SMR into multiple smaller components and draws carefully designed, simple yet powerful abstractions around them. Trantor aims at being practical, not neglecting important technical aspects such as checkpointing, state transfer, garbage collection, reconfiguration, or weighted voting, making them an integral part of the design. Trantor's modularity allows it to be flexible, maintainable, adaptable, and future-proof, which are our main design goals. Components such as the total-order broadcast protocol can easily be swapped in and out, potentially even at runtime.

Even though the focus of Trantor is not on performance, a preliminary performance evaluation of our Byzantine fault-tolerant implementation shows an attractive throughput of over 30k tx/s with a latency of under 1.3 seconds (and 0.5 seconds at 5'000 tx/s) at a moderate scale of 32 replicas dispersed over 5 different continents, despite a naive implementation of many of Trantor's components.

1 Introduction

State machine replication (SMR) is a way of achieving fault-tolerant execution of an application by replicating it across multiple connected machines – nodes of a computer network that we call replicas. Each replica executes an identical copy of the application. SMR, and in particular Byzantine fault-tolerant SMR, which tolerates arbitrary (Byzantine) faults of replicas and clients, has become a fundamental building block of blockchain systems.

We model the application as a deterministic state machine. Starting from some initial state, its state is deterministically updated by applying transactions to it. Thus, the state of the application is completely determined by the initial state and an ordered sequence of transactions known as the transaction log. It is the SMR system's task to maintain an identical copy of the ever-growing transaction log on each non-faulty replica and apply it to the application, despite the faults of some replicas. An SMR system runs across multiple replicas in a network, each initialized with an application and its initial state. It receives transactions, submitted by clients, on its input, orders them, and ensures that each replica's copy of the application executes all the transactions in the given order.

Many blockchain systems share this SMR architecture, such as Ethereum [18], Filecoin [8], The Internet Computer [3], Aptos [17], and many others. These systems are usually designed around a single consensus (transaction ordering) protocol that is built in the node software and tightly coupled with other components. The monolithic design often integrates many concepts from transaction dissemination, through ordering, all the way to execution. Swapping the execution engine for another one or completely changing the consensus protocol, with minimal changes to the rest of the system, would probably imply significant design and implementation overhead.

This paper presents Trantor, a novel SMR system. The main design goal of Trantor is high modularity, not just in its implementation, but at a conceptual level. The reason we focus on modularity is to enable tailoring an SMR system to different deployment scales, allowing for different sub-protocol implementations and catering for different application and performance needs. These goals are especially important in multi-instance SMR / blockchain deployment architectures aiming at scaling bockchain systems, such as Interplanetary Consensus (IPC) [15].

Trantor builds on the idea of separating the dissemination of transaction payloads from ordering (also used in other systems [5, 6, 9]), and takes this concept even further. It consists of multiple loosely coupled components with well-defined event-based interfaces, as is common when defining distributed abstractions. Different stages of Trantor's operation are separate components that interact through events. E.g., when a new transaction batch (also called a block) is appended to the transaction log by the ordering component, it triggers an event to which the fetching component reacts by fetching the corresponding transaction data and emitting another event that triggers the execution. In addition to making it easy to understand and independently reason about individual components, such an execution model is particularly well suited for being expressed in the Mir framework [16] which we use to implement Trantor.

Trantor delivers a continuous stream of transaction batches to the application on each replica, but its usage of memory and storage does not grow indefinitely. It is up to the application to process this (infinite) stream of transaction batches in a way that does not exhaust its resources. To this end, Trantor periodically asks the application to create a snapshot of its state after the application of a specific prefix of the transaction log. The application is expected to be able to completely restore its state from such a snapshot. Trantor includes these snapshots (along with other metadata) in checkpoints. After creating such a checkpoint, Trantor deletes all data pertaining to the checkpointed prefix of the transaction log.

The set of replicas Trantor runs on can dynamically change. Those changes are driven by the application itself, by passing the replica configuration information (consisting of identities and network addresses of the replicas) to Trantor at run time. The initial replica set is Trantor's configuration parameter. Trantor then periodically requests the next configuration from the application and parametrizes its components accordingly.

The basic interaction between Trantor and the application is similar in spirit to CometBFT's ABCI++ interface [13] and is depicted in Figure 1. We describe it in detail in Section 5.6.

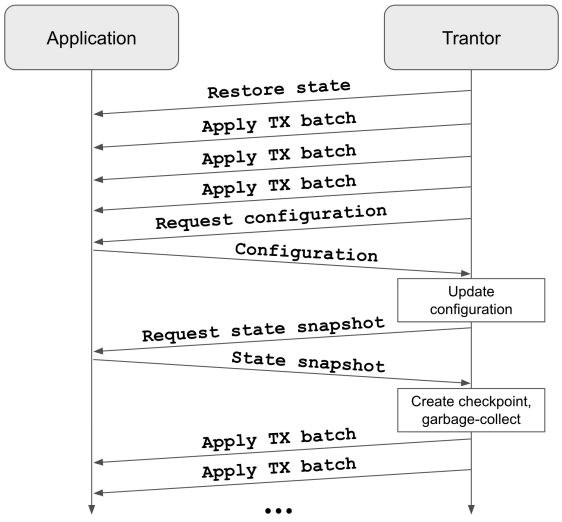

Figure 1: Interaction between Trantor and the application it replicates.

In the rest of this document, we briefly discuss related work in Section 2 and define the system model in Section 3. Section 4 presents an overview of Trantor's high-level architecture and mode of operation. In Section 5, we describe Trantor's components in more detail and discuss some other noteworthy aspects of its design. We show a preliminary performance evaluation in Section 7. Finally, we conclude in Section 8.

2 Related Work

Trantor proposes a decomposition of an SMR system at the conceptual level, and follows this decomposition in its system design. Some systems, notably Sui [12] with Bullshark [9] as their consensus engine, already successfully make the point of separating transaction payload dissemination from ordering. Hyperledger Fabric [1] also separates execution from ordering in its execute-order-validate architecture. Still, swapping different ordering engines is notoriously difficult and until today, Fabric does not seem to have a proper Byzantine fault-tolerant ordering service. Trantor takes these concepts further by treating modularity as the first priority.

CometBFT [14], powering the Cosmos ecosystem, is an SMR system similar to Trantor in its interaction with the application. Trantor's application interface follows a similar principle as CometBFT's ABCI++, more precisely, the subset of it that drives the application's state updates. However, ABCI++ also provides access to the application's state, making CometBFT a proxy for querying the application state. Moreover, CometBFT is tightly coupled with the Tendermint consensus protocol that is an integral part of the system.

3 System Model

This section describes the basic model and assumptions Trantor uses. In a nutshell, we consider an eventually (partially) synchronous message passing system consisting of replicas interconnected with reliable authenticated point-to-point links. We assume that a Byzantine adversary corrupts some of the replicas (we call those faulty) and controls the scheduling of all network messages (within the constraints of eventual synchrony), but cannot subvert standard cryptographic primitives, e.g., invert secure hash functions.

We generalize the usual constraint on the extent to which the adversary can corrupt replicas. Instead of limiting their number (using the classic "f out of n" approach), we consider each replica to be associated with a weight as part of that replica's identity. Roughly, the weight corresponds to the replica's "voting power" in the system. The total weight is the sum of the weights of all replicas. An example of such voting power that Trantor can be parameterized with is replica stake, as used in Proof-of-Stake protocols.

We define a weak quorum as any set of replicas whose combined weights exceed one third of the total weight. Analogously, a strong quorum must exceed two thirds of the total weight.[1] We assume that the adversary cannot corrupt a weak quorum of replicas. Note that, in the special case when each replica has the same weight, our model is equivalent to the adversary corrupting less than one third of the replicas.

4 Overview

4.1 High-Level Architecture

We now present a simplified view of Trantor's high-level architecture. We intentionally leave out some details and corner cases in order to convey a high-level idea of Trantor's general operation. Conceptually, Trantor can be subdivided in 5 stages:

1. **Mempool:** Receive transactions and group them into batches.

[1] We could further generalize by defining the quorum thresholds arbitrarily, as long as a weak and a strong quorum always intersect, but we stick with one third and two thirds for simplicity.

2. **Dissemination:** Make transaction batches available using consistent broadcast [2] augmented by an availability certificate. This roughly corresponds to, making a weak quorum of replicas persistently store the batches, such that any replica can fetch them in the future.

3. **Ordering:** Produce a totally ordered transaction log consisting of the availability certificates created at the dissemination stage.

4. **Fetching:** Obtain the transactions using references as they appear in the transaction log.

5. **Execution:** Apply the transactions to the application state.

Each stage is the responsibility of a separate component that we describe below. Logically, components communicate with other components by emitting events that other components react to. Note that we only use the notion of events to model the interaction between Trantor's components as is the standard in distributed system literature [2]. We do not prescribe a particular communication model for an implementation, even though Mir, the general framework for implementing distributed systems [16] that we use to implement Trantor, does directly implement such an event-based model. Alternatively (and equivalently), one can see sending an event to a component as asynchronously invoking a function of that component's interface.

Figure 2 shows a simplified diagram of Trantor's basic operation.

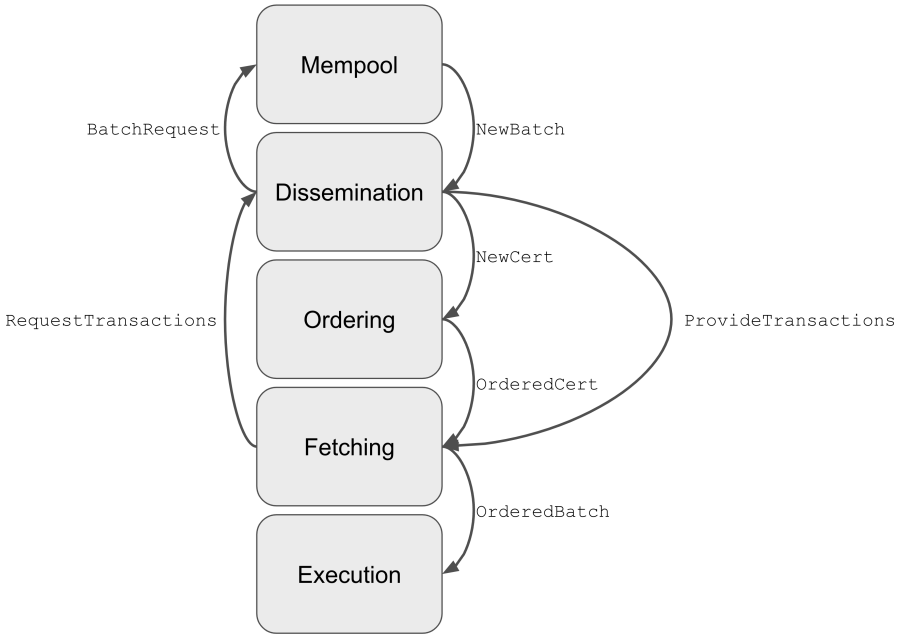

Figure 2: Trantor's basic operation.

4.1.1 Mempool

The mempool component is responsible for transaction input. It stores received transactions that are pending for ordering and execution. Similarly to the application logic, we consider the mempool an external dependency that only needs to implement an interface for interacting with

the rest of the system. Thus, how transactions are added to the mempool is out of scope of this document and, from Trantor's point of view, the transactions simply appear there.

It is important to note that each replica has its own local mempool that is not expected to be consistent with other replica's mempools. There is no guarantee that a transaction appearing in one replica's mempool will ever appear in another replica's mempool. If the same transaction does appear in multiple replicas' mempools, all these replicas will add their copy of the transaction to the blocks they disseminate and order. Trantor removes such duplicates at the fetching stage (see also Section 4.1.4). Methods for removing these duplicates early on and saving resources can be implemented in a future version of Trantor.

The basic operation of the mempool is very simple. When it receives a `BatchRequest` event, it selects all the transactions that need to be proposed and responds with a `NewBatch` event containing those transactions. The transactions within a batch are ordered with respect to each other. When a batch of transactions reaches the execution stage, the transactions will be executed in this order.

4.1.2 Dissemination

The replica's dissemination component has two tasks:

1. Transmitting locally stored transaction batches to other replicas to ensure their global availability and

2. Retrieving batches that are available, but not stored locally by the replica

For ensuring transaction availability, this component obtains transactions from the mempool and broadcasts them to other replicas. For each individual broadcast batch, the broadcast primitive implemented by the dissemination component is that of consistent broadcast, which ensures that no two correct replicas will observe a different batch (i.e., prevents even faulty replicas from equivocating). Moreover, the implementation also produces an availability certificate for each disseminated batch. An availability certificate can be verified by any other replica and serves as a proof of global availability, i.e., that the corresponding batch can be obtained by any correct replica using information found in the certificate.

In a nutshell, whenever the dissemination component is ready to disseminate a transaction batch, it emits a `BatchRequest` event to the mempool. Upon receiving a batch through the corresponding `NewBatch` event, the dissemination component executes a protocol that sends the batch to other replicas, asking them to persistently store it and return a signed acknowledgement. The resulting (multisig) availability certificate then consists of a reference to (hash of) the batch and a list of those signed acknowledgements from a weak quorum. The dissemination component then emits the certificate in the form of a `NewCert` event when requested through a `RequestCert` event (emitted by the ordering component).

The dissemination component's second task is retrieving batches based on a given availability certificate. This functionality is triggered by a `RequestTransactions` event containing a certificate. Here, the component simply downloads the missing transactions from any of the replicas that signed the certificate and emits them as a `ProvideTransactions` event. Since each availability certificate is always signed by at least one correct replica, the retrieval is always guaranteed to succeed.

4.1.3 Ordering

The ordering component takes availability certificates produced by the dissemination component and orders them into a sequence (called the transaction log) that is guaranteed to be the same on

all replicas. This is done using multiple parallel instances of the PBFT protocol [4], multiplexed together akin to the ISS protocol [11], albeit without eliminating duplicate transactions during ordering.

The ordering component consumes `NewCert` events (containing availability certificates) produced by the dissemination component, broadcasts (i.e., proposes) them using a total-order broadcast protocol (PBFT in our case), and emits `OrderedCert` events, each representing one entry in the transaction log.

4.1.4 Fetching

The fetching component transforms a sequence of availability certificates observed by consuming the `OrderedCert` events into a corresponding sequence of transaction batches and passes them on to the execution component by producing `OrderedBatch` events.

In general, a significant number of the availability certificates delivered by a replica have been proposed by other replicas, and thus the referenced transaction batches might not be present locally. The fetching component's task is to obtain all those batches from other replicas. It does so by exchanging pairs of `RequestTransactions` / `ProvideTransactions` events with the dissemination component.

The fetching component also filters out transactions that have already been included in earlier batches, such that every transaction is only passed to the execution component once (see Section 5.5).

4.1.5 Execution

The execution component contains the state and logic of the application provided by the user. It mainly consumes `OrderedBatch` events and applies the contained transactions to´ the application.

The execution component is also involved in checkpointing and configuring the set of replicas, which we discuss in Section 5.

4.2 Epoch-Based Operation and Configuration

We express the progress of the whole system through the length of the transaction log. Each ordered batch has a position in the transaction log that we call height. Note that the whole replicated state of the system can be identified by a single number – the height of the last applied transaction batch. The first batch has height 0.

4.2.1 Checkpoints

To capture the replicated state at a certain height, Trantor uses checkpoints. A checkpoint is a verifiable snapshot of the replicated state. It summarizes the application of all the ordered batches up to a certain height. We say a checkpoint is at height h if it comprises the first h batches of the log. A checkpoint completely replaces a prefix of the transaction log, i.e., a replica that obtains a checkpoint for height h (summarizing batches 0 to $h-1$) can re-initialize its state and directly continue ordering and executing the batch at height h.

A checkpoint contains a certificate that any replica can verify to confirm that the checkpointed state was indeed agreed upon. The certificate consists of a cryptographic hash of the checkpointed state signed by a strong quorum of replicas that agreed on that state.

4.2.2 Epochs

An epoch is a contiguous section of the transaction log.[2] Sometimes, very informally, we abuse the term epoch to also refer to the period of time during which this section is produced (note that Trantor does not explicitly deal with the notion of time[3]). An epoch has a well-defined length in terms of ordered batches. For example, if the first epoch's length is 4, it comprises the heights 0, 1, 2, and 3, and the second epoch starts with height 4. The corresponding transaction log is depicted in Figure 3.

A replica actively participates in the distributed protocols pertaining to only one epoch at a time. Only once it fills the whole epoch's transaction log does it advance to the next epoch. While in principle it is possible to run multiple epochs concurrently, we stick to one epoch at a time for simplicity.

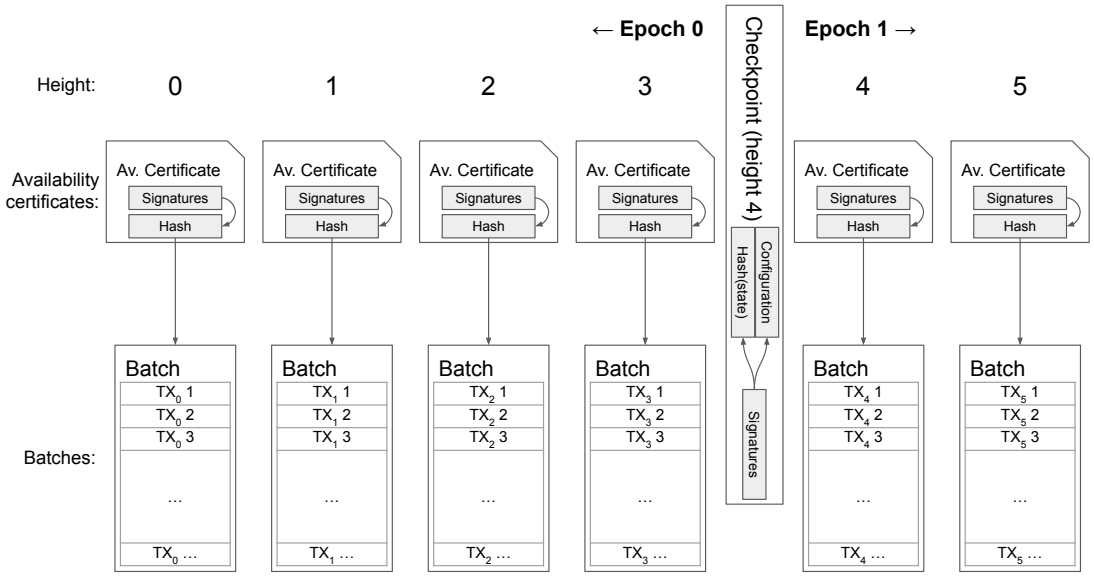

Figure 3: The very beginning of the transaction log. Depicted is the first epoch of length 4, a checkpoint and the beginning of the second epoch. Each entry in the transaction log consists of an availability certificate (signed by a quorum of replicas) that refers to a transaction batch using a hash of the batch.

4.2.3 Configuration

The configuration of the whole system (including the set of replicas) is fixed throughout an epoch, but can change from one epoch to another. At the end of an epoch, i.e., after having applied the last transaction batch to its state, Trantor requests a new configuration from the application (execution component) to use for future epochs. This configuration is part of the replicated state

[2] An epoch in Trantor is not to be confused with a Filecoin epoch, which corresponds to the notion of height in Trantor.

[3] Exposing a basic notion of agreed-upon real wall clock time to the application is a feature that can be added to a future version of Trantor.

and thus is also included in checkpoints. Trantor creates a checkpoint at the start of each epoch, summarizing the replicated state as it was at the end of the previous epoch. New replicas joining the system then only need to obtain the starting checkpoint of the first epoch the configuration of which they are part of.

To better streamline Trantor's operation, a configuration provided by the application at the end of epoch e is not used directly in epoch $e+1$, but rather in epoch $e+1+$ *configOffset*, where *configOffset* is a system parameter. A checkpoint thus contains the configuration for not just the epoch it starts, but also for *configOffset* additional subsequent epochs. Correspondingly, Trantor uses the initial configuration for the first *configOffset* $+ 1$ epochs.

4.3 Access Control and Input Validation

Trantor does not authenticate the sources of received transactions and implements no access control. We believe this is not the task of the core of an SMR system and should be implemented on top of it.

However, it is sometimes useful for the implementation of the consensus protocol to be able to validate a received proposal early (e.g., as soon as it is received from another replica), so that the consensus protocol can be aborted early for invalid proposals. Apart from opening the door to potentially important performance optimizations, such a mechanism might also be required for preventing certain forms of denial-of-service attacks.

Trantor provides the necessary tools for the application to support such early proposal validation. The ordering component supports the integration of an optional external ProposalValidator module, allowing an early abortion of the ordering of transactions that this module (as injected dependency) considers invalid.

5 Algorithm Details

This section describes the algorithms we use to implement Trantor's components in more detail and explains some additional noteworthy aspects of Trantor's design.

5.1 Dissemination

In each epoch, a subset of Trantor's replicas are designated as leaders by the ordering component (see MirBFT [10] and ISS [11] for details). Each leader replica's dissemination component continuously executes the following simple algorithm.

1. Emit a `BatchRequest` event to the mempool.
2. Wait for obtaining a transaction batch through the corresponding `NewBatch` event from the mempool.
3. Send a *`RequestSig`* message to all other replicas (including itself), containing the obtained batch.
4. Upon reception of a *`RequestSig`* message, persistently store the transaction batch and sign a digest of all its contents (our implementation uses ECDSA over the Secp256k1 curve).
5. Respond with a *`Sig`* message containing the computed signature.

6. Upon reception of valid signatures from a weak quorum (including itself), construct a multisig availability certificate out of those signatures and the batch digest, and store the certificate in a buffer. (A weak quorum of signatures ensures that there is at least one correct replica the data can be fetched from later, making the batch available.)

7. If the buffer is not yet full, restart from 1.

Upon reception of a `RequestCert` event from the ordering component, the dissemination component responds with a `NewCert` event containing all the availability certificates present in its buffer, empties the buffer, and restarts the above dissemination algorithm in case it has stopped due to a full buffer.

With this approach, many multisig certificates are contained in a single `NewCert` event. This allows for a continuous dissemination of transaction batches even before the ordering component requests any certificates. Once a `RequestCert` event arrives, the ordering component can respond immediately with the buffered certificates and the ordering can start without further delay.

The size of the buffer is a configuration parameter and also determines the maximal number of multisig certificates contained in one NewCert event. For simplicity, we refer to such a batch of certificates as a single availability certificate downstream from the dissemination component. Any operation applied to it is, under the hood, applied to all the sub-certificates. We consider an availability certificate valid iff all the contained sub-certificates are valid.

5.2 Ordering

Trantor's ordering component establishes a total order on availability certificates received from the dissemination component using a protocol that multiplexes multiple instances of PBFT. It is inspired by the ISS protocol [11], but forgoes transaction deduplication for simplicity (it can be added in a future version of Trantor).

In short, at the start of each epoch, Trantor assigns each height of that epoch to one leader replica and initializes one instance of PBFT per leader. The leader is configured to be the first PBFT primary charged with proposing availability certificates for all the heights ("sequence numbers" in PBFT terminology) assigned to it.

The liveness of each individual PBFT instance guarantees that each height will eventually be assigned an availability certificate. If the leader fails (or there is asynchrony in the network), the remaining correct replicas agree on a special empty certificate (that does not refer to any transactions) instead. Once all heights of an epoch have been assigned availability certificates, Trantor advances to the next epoch, repeating the whole process. For details on the ordering protocol, we refer the interested reader to ISS [11].

5.3 Creating Checkpoints

The creation of checkpoints is performed by a dedicated sub-protocol, but is orchestrated by the ordering component. Immediately after a transition from epoch $e-1$ to e, before any batches are appended to the transaction log, the ordering component creates a new instance of the checkpointing protocol, parametrized with a snapshot of the system state comprised of:

- The current state of the application as obtained from the execution component
- Relevant state of the Trantor protocol itself, such as epoch number, height, and other variables that need to be reinitialized when restoring the state
- Configuration of epoch e and the *configOffset* following epochs
- Information about processed transactions (needed for deduplication, see Section 5.5)

Each replica of epoch $e-1$ then executes a simple checkpointing protocol akin to the one used in PBFT:

1. Compute a digest of the snapshot and sign it.
2. Send the signed digest to all other replicas.
3. Wait for the reception of matching signed digests from a strong quorum[4] (including self).
4. Construct a (multisig) checkpoint certificate from the snapshot digest and the collected signatures.
5. The checkpoint itself consists of the state snapshot and the checkpoint certificate.

Once the checkpoint is ready, the ordering component saves it in its state, replacing any older checkpoints. Moreover, the ordering component also informs the application (execution component) about the new checkpoint. This can be useful to the application for various reasons, e.g., creating backups consistently with other replicas.

Note, however, that the constructed checkpoint certificates, all being valid, may contain different signatures on different replicas. In other words, each replica may create a different valid certificate, depending on which signatures the replica received first. In order to adhere to the paradigm of providing identical inputs to all copies of the application on all replicas, Trantor creates another instance of a consensus protocol (also PBFT in our case)[5] to agree on one single version of the certificate. This certificate is then used by all replicas for the checkpoint delivered to the application.

5.4 Garbage Collection and State Transfer

A checkpoint for epoch e (i.e, appearing between epochs $e-1$ and e in the transaction log) contains all information necessary for a replica to start / continue operating from the beginning of epoch e. Thus, as soon as such a checkpoint is available to a replica, it garbage-collects all data related to earlier epochs. The rationale is that the effect of any message that would need to be sent for any of the previous epochs is subsumed by the checkpoint which, if needed, can be transmitted instead.

Any replica that falls behind (e.g., due to transient network failures or simply due to being slow) can completely recover its state from a checkpoint. To this end, each replica keeps track of the highest epoch it observed in communication received from each other replica. When a replica r notices the epoch of another replica r' being lower by a predefined threshold (Trantor's system parameter), r sends its latest checkpoint to r'. r' does not need to wait for any quorum to confirm the validity of the checkpoint, as it can verify the received checkpoint's certificate.

Note that this naive state transfer algorithm is correct, but rather inefficient, as it leads to multiple correct replicas sending the same checkpoint (a potentially bulky data structure) to the same straggling replica. This inefficiency can easily be optimized away by only notifying the straggler about the own progress and letting the straggler request the checkpoint data from only one replica at a time (or even request chunks of it from multiple replicas). Another possibility

[4]Using a weak quorum would be sufficient under our assumptions, since it would still guarantee a correct replica storing a provably correct (thanks to the certificate) checkpoint. However, since we use checkpoints as a condition for a replica to advance epochs, requiring a strong quorum limits the number of slow replicas that can fall behind to less than a weak quorum.

[5]In our modular implementation of Trantor, we use (different instances of) the very same PBFT module to agree on both availability certificates and checkpoints - an example of the benefits of modularity.

is storing the checkpoint in an external data store available to all replicas and only transfer a reference (e.g., an IPLD pointer) to it between replicas.

Since checkpoint (multisig) certificates consist of a list of signatures, the verification relies on the knowledge of the signers' public keys. If the set of replicas (i.e., membership) never changes, all replicas can easily verify checkpoint certificates based on their knowledge of all other replica's public keys. If, however, we want to allow the system to dynamically reconfigure the membership (see Section 5.6) and, at the same time, not place any assumptions on the correctness of replicas removed from the replica set, verifying checkpoint certificates becomes more difficult due to possible long-range / "i still work here" attacks [7]. Handling this case exceeds the scope of this document and will be added to a future version of Trantor.[6]

5.5 Transaction Identification and Post-Order Deduplication

In order to apply each transaction exactly once, Trantor needs to keep track of which transactions have been applied in the past. To this end, each transaction is uniquely identified by a tuple (*ClientID*, *TxNo*) with ClientID uniquely identifying the entity that created the transaction (the client), e.g., a user of the system. The transaction number *TxNo* is a number that each client assigns to each transaction it produces. The client is responsible for producing transactions with monotonically increasing *TxNo*s. If two transactions with different payloads but the same *TxNo*s are produced by the same client, Trantor does not guarantee processing any of them. One (not both) of them, however, still might be applied.

Trantor uses these transaction identifiers at the fetching stage to filter out transactions that have already been executed previously. For each client, Trantor keeps an entry for all *TxNo*s passed to the execution component. When the fetching component obtains a batch of transactions from the dissemination component, it filters out those for which an entry already exists.

The data structures required by such a simple filter, however, would grow indefinitely and eventually exhaust all the system's resources. We therefore introduce the concept of a client watermark window - a range of *TxNo*s a client is allowed to concurrently submit to the system. If a transaction outside of the client's watermark window appears in the mempool, Trantor is allowed to ignore it. At each epoch transition, Trantor sets the start of each known client's watermark window to the lowest *TxNo* not yet executed. The window then extends over the next *WindowSize* transaction numbers, where *WindowSize* is a system parameter.

Leveraging the client watermark window, Trantor only needs a constant amount of memory to represent all transactions ever executed that originate from a single client. The total memory overhead is thus $O(c)$, c being the number of clients. Reducing this overhead even further is possible under some mild additional synchrony assumptions by garbage-collecting information about inactive clients, but is out of the scope of this document and may be implemented in a future version of Trantor.

5.6 Execution

The execution component is responsible for managing the application state and execution. Its tasks are the following.

- Apply transactions contained in ordered batches to the application logic.
- Consume periodically generated checkpoints.

[6] If Trantor is used in an IPC [15] subnet, for example, the parent subnet within the IPC hierarchy can naturally serve as a trusted source of membership information.

- Create snapshots of the application state.
- Restore the application state from snapshots.
- Dynamically configure the Trantor system (e.g., define the replica membership).

The application logic is an external dependency that Trantor's execution component simply relays events to. For convenience, Trantor supports two variants of the application logic: a simplified static one and a full-fledged one with reconfiguration support.

5.6.1 Simple Static Application

The simple static variant only needs to implement the following functions:

- `ApplyTXs(transactions)`: Applies a batch of transactions to the state machine.
- `Snapshot() → serialized_state`: Returns a snapshot of the application state.
- `RestoreState(checkpoint)`: Restores the application state from a checkpoint.
- `Checkpoint(checkpoint)`: Announces a new checkpoint to the application.

The Snapshot and RestoreState functions are tightly coupled; what Snapshot returns, RestoreState must be able to parse. The `serialized_state` value (represented as an opaque byte array) returned by Snapshot will be contained (as one of the fields) in the checkpoint that Trantor passes to RestoreState (other fields contain additional metadata such as the checkpoint certificate). The application thus has full control over serializing and deserializing its own state. This means, in particular, that a large application state need not necessarily be all encoded in the value returned by Snapshot, but can be, for example, a reference to an external data store (e.g., IPFS) that stores the actual state and to which all replicas have access.

The Checkpoint function is technically not necessary for implementing a textbook version of state machine replication and can safely be ignored by an application implementation that does not make use of checkpoints. In practice, however, this feature can be useful, e.g., for backup purposes.

5.6.2 Reconfigurable Application

Applications that wish to be able to dynamically reconfigure the Trantor system can use the full-fledged application interface. On top of the simple one, the full-fledged application interface contains one additional function:

- `NewEpoch(number) → configuration`: Announces the start of a new trantor epoch with number number. Returns a configuration to be used by Trantor after *configOffset* epochs.

We expect the decisions on new configurations to be made by the replicated application logic, based only on the replicated state. Since Trantor consistently invokes the `NewEpoch` function after having applied the same sequence of transactions on each replica, we ensure that the configuration decisions will be consistent across different copies of the application at all correct replicas as well.

5.6.3 Example Execution

To illustrate the operation of the execution component, Listing 1 shows the sequence of application function invocations as they would appear at each replica in a system with epoch length 4, corresponding to the transaction log depicted in Figure 3.

```
 1. NewEpoch(0)
 2. ApplyTXs(TX_0 1, TX_0 2, ...)
 3. ApplyTXs(TX_1 1, TX_1 2, ...)
 4. ApplyTXs(TX_2 1, TX_2 2, ...)
 5. ApplyTXs(TX_3 1, TX_3 2, ...)
 6. Snapshot()
 7. Checkpoint(checkpoint)
 8. NewEpoch(1)
 9. ApplyTXs(TX_4 1, TX_4 2, ...)
10. ApplyTXs(TX_5 1, TX_5 2, ...)
11. ...
12. ...
```

Listing 1: Application of the application corresponding to Figure 3.

The purpose of Trantor can be summarized as ensuring that exactly his sequence of application invocations happens consistently at all replicas, regardless of transactions appearing in the mempools in different orders and some replicas failing or even misbehaving.

Note that Trantor does not invoke the `RestoreState` function under normal circumstances. It is only invoked on replicas that fall behind and need to catch up, or on replicas that freshly joined the system after reconfiguration. The application must be ready to receive such an invocation at any point in time. Trantor guarantees, however, that the restoration happens at epoch boundary, i.e., Trantor only restores a snapshot taken at the end of an epoch, before any transactions from the next epoch have been applied. The standard sequence of invocations from the start of an epoch then follows (unless state needs to be restored again), as in Listing 2.

```
 1. NewEpoch(0)
 2. ApplyTXs(TX_0 1, TX_0 2, ...)
 3. ApplyTXs(TX_1 1, TX_1 2, ...)
 4. ApplyTXs(TX_2 1, TX_2 2, ...)
 5. RestoreState(checkpoint)
 6. RestoreState(checkpoint)
 7. NewEpoch(8)
 8. RestoreState(checkpoint)
 9. NewEpoch(532)
10. ApplyTXs(TX_... 1, TX_... 2, ...)
11. ApplyTXs(TX_... 1, TX_... 2, ...)
12. ...
13. ...
```

Listing 2: Restoring application state from checkpoints.

6 Implementation

We now present our implementation of Trantor, diving deeper into Trantor's architecture and refining the descriptions from Sections 4 and 5. We implement Trantor in Go, using the Mir distributed system implementation framework [16]. Note that we are describing the implementation of a *single replica*. Each replica runs its own copy of what we present in this section. In the following, we first explain some overarching concepts relevant for all modules and then describe each module in detail.

6.1 Epochs

As mentioned in Section 4.2, Trantor's operation is epoch-based. Therefore, several components are, explicitly or implicitly, associated with a concrete epoch. The Orchestrator module drives progress from one epoch into the next one. It instantiates and initializes all the modules necessary for an epoch, assembles the corresponding part of the event log from their output, triggers the creation of a checkpoint, and, when the checkpoint is available, triggers garbage collection.

Several modules in Figure 4 (Dissemination, Ordering, Checkpointing) contain submodules. While the top-level modules are long-lived and exist throughout Trantor's operation, the submodules are generally only relevant for one epoch, for which the Orchestrator dynamically instantiates them, and are eventually garbage-collected.

6.2 Garbage Collection Using the Retention Index

In Trantor, garbage collection is very explicit. Each data structure (including whole submodules) that is subject to garbage collection is explicitly associated with an integer *retention index*. The retention index is a system-wide, monotonically increasing variable driven by the Orchestrator. All data associated with a retention index lower than the current one is defined to be obsolete and safe to delete. In Trantor, the retention index is derived from the epoch number, resulting in the garbage collection semantics described in Section 5.4.

We could have performed garbage collection directly based on the epoch number, but using the retention index helps keep the semantically different concepts "progress in SMR" and "garbage collection" cleanly separated. This way, modules that otherwise do not have a reason to be aware of the progress of the whole state machine (or even of the concept of SMR) – which are, in fact, most of the modules – can be cleaner, simpler, more general, and thus more reusable.

6.3 Trantor's Modules

Figure 4 displays all relevant modules of Trantor's implementation and their interactions. It rather accurately represents the structure of our code, omitting only few technical details not relevant for understanding the workings of the implementation (e.g., the mechanism for verification of certificates or computing unique identifiers of transactions and transaction batches). To improve the presentation, the names of some events and modules also do not necessarily fully correspond to those used in our code.[7]

For simplicity, this description also does not distinguish between modules handling local events (received from other modules) and network messages (from other replicas), even though they are handled slightly differently in practice, since, unlike local events, network messages must always be assumed to be potentially created by a malicious adversary.

[7] We will be consolidating our implementation of Trantor to closely match the more intuitive terminology used in this document, rather than the other way round (using less intuitive terminology in this document just to match our implementation).

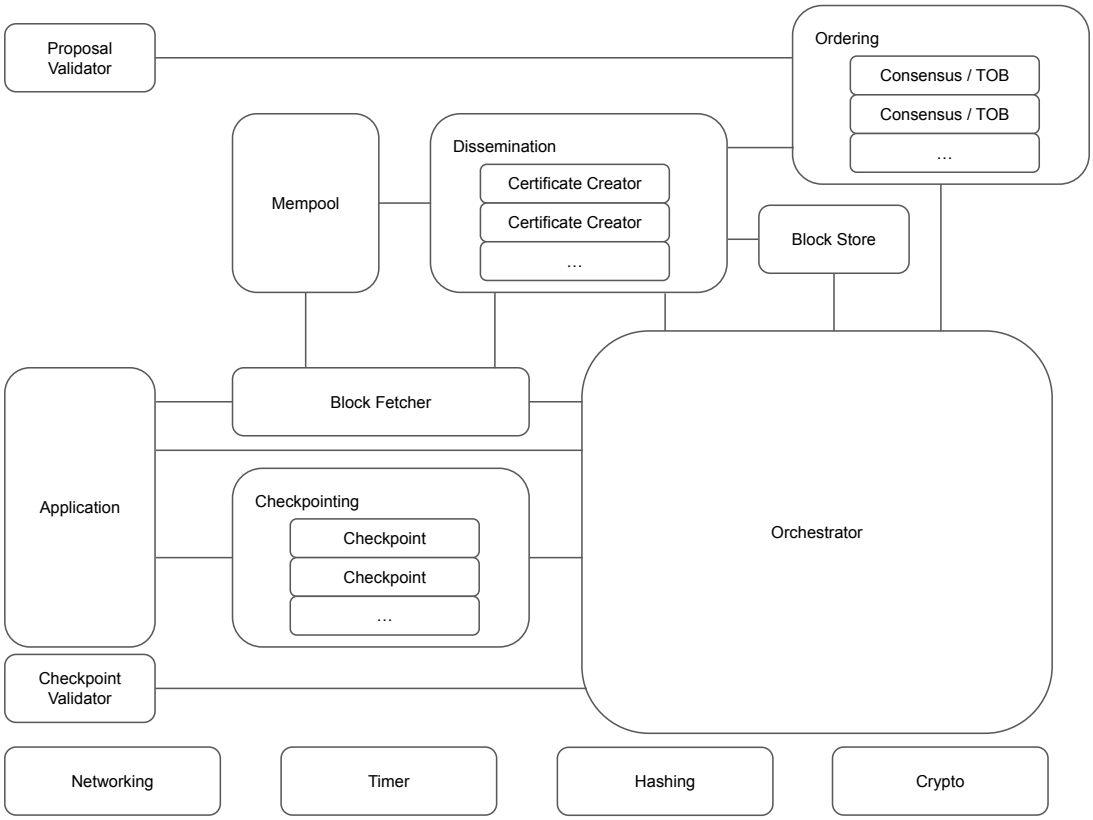

Figure 4: Schematic view of Trantor's implementation's components. A line between two boxes means that the two components interact. The four bottom-most components (Networkig, Timer, Hashing, and Crypto) have a supporting role. They interact with many other components and we omit the corresponding lines in the diagram for simplicity of presentation.

6.3.1 Orchestrator

The Orchestrator is the heart (and brain) of Trantor. Its main task is to advance the transaction log. In addition to the transaction log data structure, the Orchestrator maintains the configurations (including replica membership information) of the current and *ConfigOffset* next epochs, as well as the most recent checkpoint. It also keeps track of the progress of other replicas in order to decide when they need to be sent a checkpoint to catch up.

Trantor always starts operating from an initial checkpoint (equivalent to a genesis block in some other blockchain systems), with which the Orchestrator is initialized. A checkpoint, by definition, defines the system state at the start of an epoch. The Orchestrator thus initializes all its internal data structures and emits events to other modules corresponding to the start of a new epoch. This includes instructing the Dissemination and Ordering modules to create new submodules executing, respectively, the dissemination and ordering protocols for the new epoch. The Orchestrator then waits for events from the Ordering module announcing the delivery of availability certificates for all the heights of the epoch, which it relays (in the correct order) to the Block Fetcher. Each delivered availability certificate is explicitly annotated with its position in the transaction log and the Orchestrator does not expect the certificates to be delivered in

order.

At any point in time, the Orchestrator may receive a `RestoreFromCheckpoint` event, in which case it performs a re-initialization as described above, using the received checkpoint.

The Orchestrator consumes the following events:

- `DeliverCert(height, cert)`
  This event means that one of the Consensus/TOB modules delivered an availability certificate for some particular height. The Orchestrator inserts the delivered availability certificate at the corresponding height in the transaction log. The Orchestrator then announces as many availability certificates from the transaction log as possible to the Block Fetcher, while respecting in-order delivery (i.e., announcing the availability certificates according to their heights).

- `NewCheckpoint(checkpoint)`
  This event means that the Checkpoint module produced a checkpoint. In turn, the Orchestrator initiates agreement on this new checkpoint (see Section 5.3). To this end, it creates a new instance of a Consensus/TOB module – more precisely, makes the Ordering module do so – by emitting a `NewModule` event to Ordering, with the new checkpoint set as the proposal.

- `DeliverCheckpoint(checkpoint)`
  This event means that the Consensus/TOB module agreeing on the checkpoint reached a decision. The Orchestrator updates its most recent checkpoint and announces the agreed-upon checkpoint to the Block Fetcher.

  Moreover, the Orchestrator performs garbage collection of both its internal state and several other modules (such as Dissemination, Ordering, and others) by triggering their corresponding `GarbageCollect` events with a retention index that corresponds to the epoch of the delivered checkpoint.

- `EpochProgress(replica, epoch)`
  This event notifies the Orchestrator about a replica having advanced to a particular epoch. The Orchestrator updates its local view about that replicas progress (the maximal epoch it has been seen advancing to). The Orchestrator uses this information when deciding which other replicas need to be sent a checkpoint to restore their state from. This event is triggered by the Checkpoint module whenever it receives a message from a replica. While other modules could also trigger this event on the reception of an epoch-annotated message, we chose the Checkpoint module, as it only receives one message per epoch per replica.

- `RestoreFromCheckpoint(checkpoint)`
  When another replica considers this replica to have fallen behind (by the mechanism explained just above), the Orchestrator will receive the `RestoreFromCheckpoint` event (triggered by a message from the other replica). If the received checkpoint is indeed more recent than the locally stored latest checkpoint, the Orchestrator initiates validation of the received checkpoint by triggering a `ValidateCheckpoint` at the Checkpoint Validator.

- `CheckpointValidated`
  When the Checkpoint Validator successfully validated a received checkpoint, the Orchestrator re-initializes the whole system from it, as described above.

- `PushCheckpointTimeout`
  The Orchestrator periodically receives this event from the Timer module. Each time, it

checks the observed progress of other replicas and, if an of them is locally perceived as having fallen behind, sends sends them the latest checkpoint to restore their state from.

- `NewConfig`
  As described in Section 4.2.3, the Application module defines the system configurations. Those have the form of a `NewConfig` event received by the Orchestrator, which stores the configuration until until the epoch in which it needs to be applied.

6.3.2 Proposal Validator

If Trantor is configured to use external input validation (see Section 4.3), the Consensus/TOB module may request the Proposal Validator to validate consensus proposals received from other replicas. The Proposal Validator's implementation can perform arbitrary (potentially application-specific) checks on the proposal and notifies the Consensus/TOB about their result.

The Proposal Validator module consumes only one type of event:

- `ValidateProposal(proposal)`
  The Proposal Validator checks the validity of the given proposal and emits a `ProposalValidated` event back to the module that emitted the `ValidateProposal` event (the Consensus/TOB module in our case). Note that nothing prevents the Proposal Validator from communicating with other modules (in particular, with the Application), or even with other replicas before responding to the validation request.

6.3.3 Ordering

The Ordering module is responsible for managing Consensus/TOB submodules. In each epoch, it contains one or more submodules that agree on the assignment of availability certificates to different heights in the transaction log. Moreover, in each epoch, Ordering contains one more submodule to agree on the epoch checkpoint. The Ordering module creates and destroys the submodules as commanded by the Orchestrator.

The Ordering module consumes the following events:

- `NewModule(moduleConfig, retentionIndex)`
  Upon receiving this event, the Ordering module instantiates a new Consensus/TOB submodule (that implements the PBFT protocol in our case). The created submodule is associated with the given retention index and parametrized according to `moduleConfig`. Parameters include the membership, the heights for which the submodule needs to reach agreement, an optional proposal for each height, and some protocol-specific parameters.[8]

- `GarbageCollect(retentionIndex)`
  Upon reception of this event (from the Orchestrator), the Ordering module deletes all its submodules that have been created with a lower retention index than the given one.

6.3.4 Consensus/TOB

The Consensus/TOB modules are dynamically created and deleted by the Ordering module and are responsible for executing the consensus / TOB protocol. A Consensus/TOB module is oblivious of the epoch it is part of or the other Consensus/TOB instances. It simply receives a list of

[8] Even the protocol itself could be chosen by a parameter, even though our implementation only supports PBFT so far.

heights as part of its configuration, and for each of these heights, it needs to agree on a value and announce it to the Orchestrator by emitting a `Deliver` event.

If the Consensus/TOB module is configured with a proposal, it will use that proposal as input for the consensus protocol. Otherwise, it will request an availability certificate from a Disseminator module it is configured to use and proposes that certificate.

The Consensus/TOB module consumes the following events:

- `NewCert(cert)`
  In case the Consensus/TOB module has not been pre-configured with a proposal and requested a new certificate from the Disseminator, the `NewCert` event carries a new availability certificate that the Consensus/TOB module uses as a consensus proposal.

- `ProposalValidated(validationResult)`
  This event is relevant only if the Consensus/TOB module's implementation relies on external proposal validation and has emitted a `ValidateProposal` (see Section 6.3.2) event. If so, the `ProposalValidated` event carries the result of the validation and triggers whatever further processing of the consensus protocol that has been waiting for it.

- The other events consumed by the Consensus/TOB module are specific to its implementation. In our case, those are events for receiving various protocol message (`Preprepare`, `Prepare`, `Commit`, `ViewChange`, etc.) and various PBFT-specific timeouts.

6.3.5 Dissemination

Similarly to the Ordering module managing instances of Consensus/TOB, the Dissemination module manages instances of Disseminator submodules. The main difference is that Trantor only uses a single Disseminator per epoch.

The Dissemination module consumes the following events:

- `NewModule(moduleConfig, retentionIndex)`
  Upon receiving this event, the Dissemination module instantiates a new Disseminator submodule. The created submodule is associated with the given retention index and parametrized according to `moduleConfig`.

- `GarbageCollect(retentionIndex)`
  Upon reception of this event (from the Orchestrator), the Dissemination module deletes all its submodules that have been created with a lower retention index than the given one.

6.3.6 Disseminator

Disseminators are dynamically created and deleted by the Dissemination module and are responsible for disseminating transaction payloads to other replicas (see Section 5.1) and producing availability certificates. Whenever a Disseminator produces a new availability certificate, it emits a `NewCert` event.

A Disseminator is oblivious of the epoch it is part of and simply executes the dissemination algorithm according to the parameters it has been given.

The Disseminator consumes the following events:

- `RequestCert`
  This event requests a new availability certificate. In response, the Disseminator triggers a `NewCert` event containing the produced certificate. Note that this need not necessarily happen immediately. The implementation of the Disseminator may only trigger the data

dissemination upon reception of a `CertRequest`. In our implementation, however, the Disseminator continuously disseminates transaction data and buffers the resulting certificates that it can immediately use as a response to a `CertRequest`

- `NewBatch(batch)`
  Through this event, the Mempool announces a new transaction batch ready to be disseminated. The Mempool emits the `NewBatch` event as a response to a `RequestBatch` previously emitted by the Disseminator.

- `BatchStored`
  The `BatchStored` event notifies the Disseminator that a batch has been persisted in the Block Store. This occurs when the Disseminator has previously asked (through a `StoreBatch` event) the Block Store to store a batch during the dissemination process. In our implementation, this is when the Disseminator signs the stored batch and sends a `Sig` message (see Section 5.1).

- `RequestTransactions(cert)`
  Upon reception of this event, the Disseminator looks up all the transactions (including their payloads) referenced by `cert`. If the referenced transactions are not present locally (in the Block Store), the Disseminator requests the missing transactions from replicas that have signed `cert`. The Disseminator then and emits a `ProvideTransactions` event containing those transactions.

- `BatchLookupResult`
  This event informs the Disseminator of the result of looking up a transaction batch in the Block Store.

6.3.7 Block Store

The Block Store is a simple module that persistently[9] stores transaction batches (including payloads). It has a simple key-value-store-like interface and consumes the following events:

- `StoreBatch(batchID, batchData, retentionIndex)` The Block Store persistently stores `batchData` under `batchID`. The `batchID` is derived from a hash of `batchData`. The stored batch is associated with the given `retentionIndex` for garbage collection purposes. Once the batch is stored, the Block Store emits a `BatchStored` event.

- `LookUpBatch(batchID)` The Block Store looks up the given `batchID` in its local database and emits a `BatchLookupResult` event containing either the corresponding batch data or a flag that the batch was not found.

- `GarbageCollect(retentionIndex)` This event makes the Block Store module delete all batches associated with a retention index smaller than `retentionIndex`. The Block Store receives this event from the Orchestrator each time a new checkpoint is created.

6.3.8 Mempool

While, as mentioned in Section 4.1.1, the design of Trantor treats the mempool as an external dependency, our implementation includes a Mempool module that accepts `NewTransactions` events and stores the contained transactions in memory. Upon receiving a `RequestBatch` event,

[9] Our Go implementation uses an in-memory stub for the Block Store.

the Mempool responds with a `NewBatch` event containing the stored transactions. The Mempool's operation is influenced by several parameters[10]:

- *MaxTransactionsInBatch*: An emitted `NewBatch` event never contains more than *MaxTransactionsInBatch* transactions. If the number of stored transactions exceeds this value, the next emitted batch will contain the *MaxTransactionsInBatch* least recently added transactions.
- *MaxPayloadInBatch*: Analogously to *MaxTransactionsInBatch*, *MaxPayloadInBatch* defines the maximal total cumulative payload size of all transactions in a batch.
- *BatchTimeout*: To enable effective batching of transactions, the mempool does not respond to a `BatchRequest` immediately if there are not enough pending transactions for reaching at least one of the two above limits. *BatchTimeout* defines the maximal time the mempool waits for a batch to fill. After *BatchTimeout* elapses, the mempool emits a `NewBatch`, even if it is not full (or is even completely empty).

When the Mempool emits a transaction batch, the batch still needs to be proposed, ordered, and executed. If the transactions end up not being executed for some reason (e.g., the ordering fails due to network asynchrony), Trantor needs to propose a batch with those transactions again. Thus, transactions can be removed from the Mempool only after they have been executed and the corresponding state of Trantor has been captured in a checkpoint.

At the same time, transactions that *are going to be* executed (and checkpointed) should not be included in new batches emitted by the Mempool. However, the Mempool cannot know which of the already emitted transactions are going to make it into a checkpoint and which are not. A simple solution would be to simply delay emitting a new batch until the previous one has either succeeded or failed to be executed. This is very restrictive and may be detrimental to performance. Instead, we make the Mempool aware of Trantor epochs and allow it to emit each transaction only once per epoch. We leverage the fact that epoch transitions are a form of synchronization points with no "in-flight" batches. At the end of an epoch, all executed transactions are checkpointed and can be safely deleted from the Mempool. Transactions that have not been executed need to be proposed again in the next epoch.

The Mempool also keeps track of client watermarks (see Section 5.5) and ignores all transactions that are outside of their respective clients' watermark windows. A client that has never submitted a transaction is defined to have a watermark window starting at *TxNo* 0 (no explicit "registration" of clients is necessary). Trantor updates the Mempool's watermark information at the beginning of every epoch.

The Mempool consumes the following events:

- `NewTransactions(transactions)`
  The Mempool stores all the provided `transactions` that are within their respective client's watermark windows.
- `RequestBatch` The Mempool, according to its parametrization, eventually emits a `NewBatch` event.
- `BatchTimeout` When waiting for a batch to fill, the Mempool sets up a timer (by emitting a `TimerDelay` event to the Timer module) to receive this event after *BatchTimeout*. Upon

[10]Other modules, not just the Mempool, also can be parametrized in various ways, but we omit the description of these parameters for simplicity. We make an exception for the *Mempool* and do describe its parameters, since they are relevant for our performance evaluation presented in Section 7.

reception of this event, the Mempool emits a `NewBatch` event with the corresponding batch, unless it has already done so.

- `NewEpoch(executedTXs)` The Mempool receives this event at the start of every epoch to
  - update its view of the client watermark windows according to `executedTXs`,
  - delete all `executedTXs`, and
  - reset the internal trackers of already emitted transactions.

6.3.9 Application

The Application module simply wraps the user-defined application (see Section 5.6) in a module. On reception of an event, the Application module simply calls the corresponding function of the execution component and emits an event with that function's return value (if any).

The Application module consumes the following events:

- `OrderedBatch(batch)` The Application module calls the `ApplyTXs(transactions)` function of the application.
- `SnapshotRequest` The Application module calls the `Snapshot()` function of the application and emits a `Snapshot` event containing its return value. This `Snapshot` event goes directly to the Checkpoint module responsible for creating the checkpoint the snapshot is part of.
- `RestoreState(checkpoint)` The Application module calls the `RestoreState(checkpoint)` function of the application.
- `NewEpoch` The Application module calls the `NewEpoch` function on the application and emits a `NewConfig` event containing the returned configuration.
- `Checkpoint` The Application module calls the `Checkpoint(checkpoint)` function on the application.

6.3.10 Block Fetcher

In general, it is mostly the Orchestrator that assembles a sequence of events to be consumed by the Application. This includes events for requesting state snapshots, restoring the state, and notifications about new checkpoints and epochs. It is only the `OrderedBatch` events that are missing in this sequence – the Orchestrator emits `OrderedCert`s in their place.

The Block Fetcher's main task is to replace the `OrderedCert` events in this sequence by the corresponding `OrderedBatch` events and deliver the sequence to the Application. When constructing the `OrderedBatch` events, the Block Fetcher also removes duplicate transactions as described in Section 5.5.

Replacing an `OrderedCert` by an `OrderedBatch` involves communication with the corresponding Disseminator which, in turn, might even need to communicate with other replicas over the network. In the meantime, the Block Fetcher might receive more events from the Orchestrator. The Block Fetcher can buffer such events and delay their processing until it has finished fetching the batch. This is required if the content of the batch can change the Block Fetcher's state (e.g., the information about delivered transactions) on which the processing of subsequent events depends.

The Block Fetcher keeps track of the current epoch by observing the `NewEpoch` events emitted by the Orchestrator. It uses the epoch information to always request the transaction batches from the correct Disseminator instance.

The Block Fetcher consumes the following events:

- `OrderedCert(cert)`
  The Block Fetcher requests the transaction batch from the current epoch's Disseminator instance by emitting a `RequestTransactions` event.

- `ProvideTransactions(transactions)`
  This event is the Disseminator's response to the previously emitted `RequestTransactions` event. Upon receiving it, the Block Fetcher assembles a new batch from only those `transactions` that have not yet been delivered to the Application and creates a new `OrderedBatch` event. The Block Fetcher then either directly emits the new `OrderedBatch` event or, in case a previous event is still waiting to be emitted, enqueues it for later in-order emission.

- `SnapshotRequest` This event is emitted by the Orchestrator during the creation of a checkpoint. The Block Fetcher relays this event to the Application. Moreover, since the Block Fetcher also contains state than must be checkpointed (the client watermark information), it serializes the relevant part of its state and emits a `ClientProgress` event directly to the corresponding Checkpoint module.

- `RestoreState(checkpoint)` The Block Fetcher relays this event to the Application and re-initializes the relevant parts of its own state (epoch number and client watermark information).

- `NewEpoch` The Block Fetcher updates its local view of the epoch number and relays the event to the Application. It also emits a `NewEpoch` event to the Mempool with information about the transactions executed so far (see Section 6.3.8).

- `Checkpoint` The Block Fetcher relays this event to the Application.

6.3.11 Checkpointing

Analogously to the Dissemination module (Section 6.3.5), the Checkpointing module manages instances of the Checkpoint submodule, each of which is responsible for creating a single checkpoint. At the start of each epoch, the Orchestrator creates a new Checkpoint instance that assembles the starting checkpoint of that epoch.

The Checkpointing module consumes the following events:

- `NewModule(moduleConfig, retentionIndex)`
  Upon receiving this event, the Checkpointing module instantiates a new Checkpoint submodule. The created submodule is associated with the given retention index and parametrized according to `moduleConfig`.

- `GarbageCollect(retentionIndex)`
  Upon reception of this event, the Checkpointing module deletes all its submodules that have been created with a lower retention index than the given one.

6.3.12 Checkpoint

The Checkpoint module represents a single instance of the checkpointing subprotocol described in Section 5.3. Trantor creates one such instance at the start of each epoch. Every instance creates a checkpoint corresponding to the state of the system after finishing the previous epoch and before delivering any transactions in the next current one.

6.3.13 Checkpoint Validator

6.3.14 Networking

6.3.15 Timer

6.3.16 Hashing

6.3.17 Crypto

6.4 TODO

- Dynamic module creation, message buffering
- Multi-threaded sub-module execution

7 Performance Evaluation

This section presents a preliminary performance evaluation of Trantor, measuring the latency and throughput of ordering transactions. The purpose is not to improve upon the state of the art in any particular metric, but rather show that Trantor can provide more than sufficient throughput (tens of thousands oft transactions per second) and acceptable latency (sub-second in a global deployment) for many real-world applications. Most importantly, we show that the performance price for modularity and clean abstractions is worth paying.

For now, we only evaluate the "good-case" performance, that is, we do not artificially introduce faults or simulate Byzantine behavior of some replicas. Mechanisms for dealing with such situations are in place, however (PBFT view change), and we plan on evaluating Trantor's performance under adverse conditions in the future.

7.1 Experimental Setup

We measure Trantor's performance at a moderate scale of 32 and 64 replicas. For each replica, we use an Amazon EC2 virtual machine of type `t4g.xlarge` with 4 virtual Arm-based AWS Graviton2 CPUs and 16GB of memory (even though most of the memory is not used). The replicas are either all connected by a local network (LAN) or dispersed over the globe as far apart as possible, communicating over the internet (WAN):

- **LAN:** Single Amazon datacenter in Ireland.
- **WAN:** 8 Amazon datacenters: N. Virginia (`us-east-1`), N. California (`us-west-1`), Seoul (`ap-northeast-2`), Sydney (`ap-southeast-2`), Central Canada (`ca-central-1`), Frankfurt (`eu-central-1`), Ireland (`eu-west-1`), and Sao Paulo (`sa-east-1`).

For the WAN deployments, replicas are uniformly distributed among the 8 different locations, i.e., 4 replicas and 8 replicas in each location respectively for deployments of 32 and 64 replicas.

7.2 Load Generation Generation and Data Collection

To generate the load, we implement simple clients that generate transactions and insert them into the replicas' mempools. To measure purely the throughput and latency of transaction ordering, we co-locate each client with a replica on the same machine. In fact, each client is merely a thread within the replica's process.[11]

[11] Trantor's design is very well-suited to implement such clients – one merely needs to add a module continuously producing `NewTransaction` events consumed by the mempool.

The clients operate in a "closed loop". Each client submits a transaction with 512 bytes of randomly generated payload, waits until the transaction is delivered, and then immediately submits the next one. We consider a client's transaction tx delivered when the replica the client is located on invokes `ApplyTX(transactions)` with $tx \in$ `transactions`. In our experiments, we gradually increase the number of such clients until the system becomes saturated and the throughput stops increasing. Most of the time, it takes tens of thousands of clients to saturate the system.

Our mempool implementation is parametrized with $MaxTransactionsInBatch = 2048$ (chosen empirically after exploring a few others), $MaxPayloadInBatch =$ 16MB (a very large value that effectively disables this limit), and a varying *BatchTimeout*. The duration of each run (corresponding to one data point in the plots) is 2 minutes.

7.3 Latency Metrics

This preliminary evaluation focuses on the expected "good-case" scenario that users might expect during normal operation of the system. We conduct our experiments in a simple way, gathering the data during the whole 2 minutes of each experiment's duration. As is common in experiments where not all replicas can start up at exactly the same time, the data from the first and last few seconds are not representative of the steady state. Nevertheless, we include this data for simplicity.

Instead, we report on the median transaction latency as opposed to the common average or tail latency, as the median is not affected by the above. We thus believe it is the most representative of the normal-case steady-state operation of Trantor. Nevertheless, the difference between our measured median and average latencies small in most of the cases (within 50%). The tail (95th percentile) latency is rarely more than twice the average.

Since each client is co-located with a replica and only submits its transactions to that single replica, we really only measure Trantor's ordering overhead, not the overall end-to-end latency of a real-world application built on Trantor. Also, the delivery of a transaction by a replica that proposed it (when we measure latency) does not guarantee that the same transaction has already been delivered by other replicas as well.

In some real-world applications, where the client is remote and does not trust any single replica, the end-to-end latency would increase by 1) the time until sufficiently many replicas deliver the transaction and 2) the communication delay between the client and the slowest of these replicas.

7.4 Results

Figure 5 shows Trantor's performance when deployed across the globe. For 32 replicas, Trantor achieves a sub-second latency for throughputs of up to 10'000 tx/s, and under a load of up to 8'000 concurrent clients (throughput of 4000' tx/s), Trantor's median ordering latency is as low as 0.5 seconds. Trantor keeps a latency under 1.3 seconds until it reaches its peak throughput of above 30'000 tx/s with around 50'000 concurrent clients. At this point, the system saturates and an increased number of clients only results in an increased latency. When increasing the number of clients even further, even the throughput starts decreasing due to thrashing effects. We chose the *BatchTimeout* of 0.4 seconds empirically, as it provided the best results after a brief (and non-exhaustive) search of the parameter space.

A WAN deployment of 64 replicas yields slightly higher latency and, interestingly, a comparable peak throughput. This suggests that the network bandwidth is not the bottleneck, at least

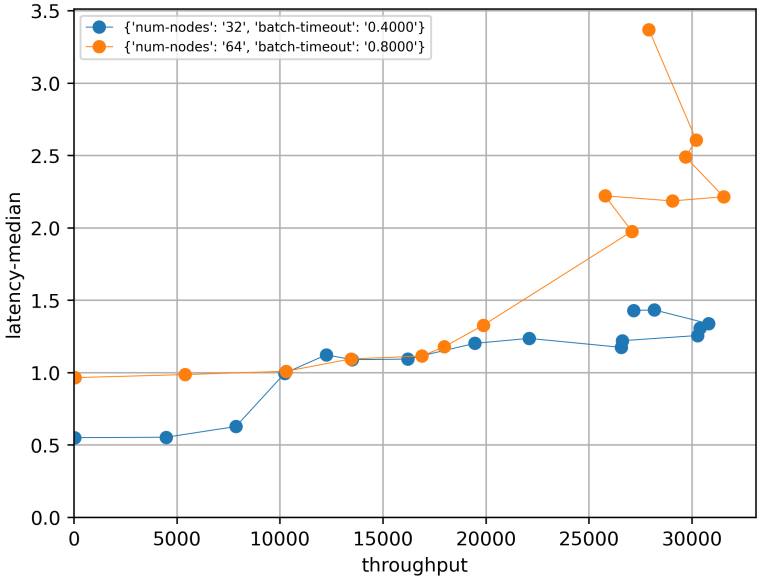

Figure 5: Median latency (in seconds) vs. throughput (in tx/s) in a WAN deployment of 32 and 64 replicas.

for the 32-replica deployment. Indeed, The average bandwidth used per replica at saturation is 16 MiB/s and 21 MiB/s respectively for the small and large scale deployment.

Another interesting phenomenon occurs at throughputs between 10'000 and ca. 17'000 tx/s, where both system sizes exhibit the same latencies. We attribute this to the different setting of the *BatchTimeout* parameter for the two different system sizes, as we increase the *BatchTimeout* at larger scale to 0.8 s to compensate for the higher message processing overhead.[12]

Figure 6 shows Trantor's performance when deployed in a single datacenter with a fast network. For 32 replicas, until the system starts becoming saturated at 60'000 tx/s, Trantor's latency is mostly under 0.25 seconds. For a deployment of 64 replicas, throughput drops to 40'000 tx/s and latency increases to just below 1 second. This is because each replica needs to wait for a larger quorum in order to make progress in both the dissemination and the ordering protocol. Also, the overhead of ordering every transaction is higher, as each replica needs to send more outgoing messages and process more incoming ones. This has a strong impact both on the network bandwidth (mostly for the dissemination component) and CPU load (mostly the ordering component). We partially compensate for the higher CPU load by increasing the *BatchTimeout* to 0.8 seconds, but it does not seem to be sufficient on a LAN. We attribute this to a bandwidth bottleneck since, at saturation, both the small and large scale deployments rarely exceed an average network transmission rate of 35 MiB/s per replica.

[12]Due to the rather large batch size limit of 2048 transactions, most of the batches in our experiments are not completely full and are triggered by the `BatchTimeout`, especially before the system is saturated.

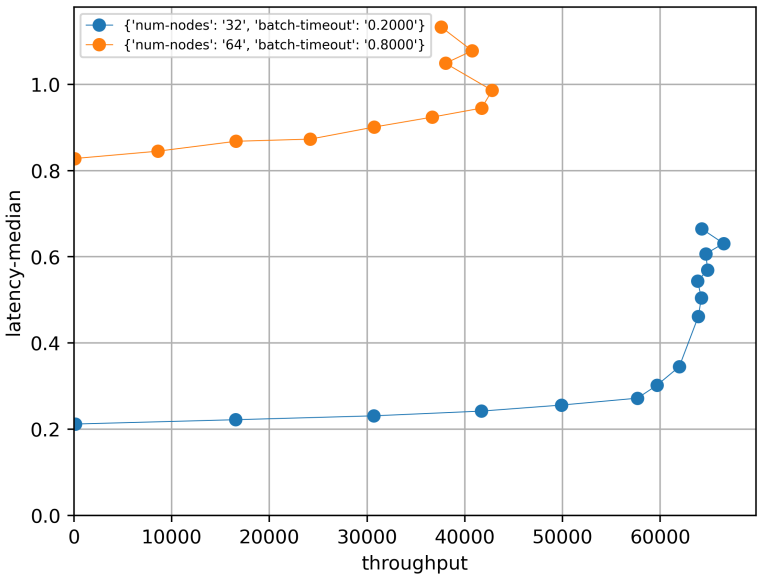

Figure 6: Median latency (in seconds) vs. throughput (in tx/s) in a LAN deployment of 32 and 64 replicas

7.5 Discussion

In the following, we discuss some additional noteworthy aspects of Trantor's performance evaluation.

LAN Latency. When deployed in a single datacenter, Trantor's latency is higher than one might expect from a system deployed on such a fast network. In fact, for 64 replicas, the LAN latency is almost as high as in the WAN. This has two main reasons.

First, the algorithms used are not latency-optimal. The very rudimentary implementation of the dissemination component adds an extra round-trip (quorum gathering) to the latency of PBFT. This can, however, be alleviated by using better protocols for dissemination and ordering, e.g., as is done in Bullshark [9].

Second, Trantor's current implementation focuses on feature-completeness, not on processing performance. Therefore, it is very likely that local message processing also has a non-negligible contribution to overall transaction latency.

Transaction Deduplication. In a scenario like the above, a client might submit the same transaction to multiple replicas' mempools. As described in Section 5.5, the current implementation of Trnator would treat them as separate transactions, lowering the effective throughput. In our experiments, such duplication does not occur, since we also do not expect it to occur during normal operation in a real-world deployment. Nevertheless, a slightly modified version of Trnator's dissemination and ordering components can easily implement a deduplication mechanism [10, 11] to mitigate this issue.

Persistent Storage. Our implementation of the Block Store is a stub that does not persistently store any data. Instead, we use a simple in-memory hash map to store the batches. The overhead of accessing stable storage is thus not taken into account in our performance evaluation.

Access Control and Input Validation. As discussed in Section 4.3, Trantor does not, by default, perform access control or input validation (although it can be configured to do so). Thus, our evaluation does not take into account it's potential overhead.

Unoptimized Implementation. Trantor's current implementation focuses on modularity and feature-completeness rather than on immediate performance optimization. We prefer having smaller, well-encapsulated modules with simple yet meaningful interfaces that significantly simplify reasoning about both their interaction and implementation. For example, fetching missing transactions has a very simple interface (`RequestTransactions`/`ProvideTransactions`), but its implementation is the most naive possible: requesting all transactions referenced by the certificate from all replicas that signed it, introducing manyfold redundancy.

Our Go implementation also uses (for historical reasons) an extremely inefficient representation of messages based on Protocol Buffers, where simple deserialization of a received message involves more than 10 (in some cases potentially more than 20) memory allocations.

Trantor achieves the presented performance despite these and many more inefficiencies. More importantly, Trantor makes it easy to optimize each component separately in the future. Even this document mentions several improvements that can be implemented in a future version of Trantor. Trantor's architecture makes it easy to do so with only minimal modifications (if any at all) to the present system. If deemed necessary, even optimizations that intrinsically require breaking abstraction boundaries can be implemented in Trantor by replacing multiple involved components by a single, more integrated one.

8 Conclusions

Trantor is a state machine replication (SMR) system that allows an arbitrary application modeled as a deterministic state machine to be deployed across multiple replicas in a fault-tolerant way. Trantor's design focuses on modularity. We decompose the problem of SMR at the conceptual level into smaller sub-problems, each of which can be studied, reasoned about, and implemented separately. We design the solution to each of these sub-problems as a separate component of Trantor. In the future, it should be easy to replace or rearrange these components to adapt the system to varying requirements and new state of the art.

A preliminary good-case performance evaluation of Trnator shows very promising results with a throughput in the tens of thousands tx/s and sub-second latency, even when deployed over a WAN across the globe. All this despite its simple implementation that still leaves a very large potential for performance optimization.

Trantor does not claim novelty in terms of protocols used or performance achieved. In fact, the implementation of some of Trantor's components is naive and sub-optimal compared to what is achievable in theory (and in practice). It is not (yet) the point of Trantor to employ the most state-of-the-art component implementations, but to enable anyone to do so easily. The ultimate goal is to be able to cherry-pick existing solutions to the multitude of sub-problems in SMR and flexibly combine them within one SMR system.

References

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *EuroSys*, 2018.

[2] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2011.

[3] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. Cryptology ePrint Archive, Paper 2021/632, 2021. `https://eprint.iacr.org/2021/632`.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

[5] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Proof of availability and retrieval in a modular blockchain architecture. Cryptology ePrint Archive, Paper 2022/455, 2022. `https://eprint.iacr.org/2022/455`.

[6] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *EuroSys*, 2022.

[7] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 2019.

[8] Protocol Labs. Ethereum: A secure decentralised generalised transaction ledger. `https://filecoin.io/filecoin.pdf`, 2017.

[9] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *CCS*, 2022.

[10] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolic. Mir-bft: Scalable and robust BFT for decentralized networks. *J. Syst. Res.*, 2022.

[11] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *EuroSys*, 2022.

[12] The MystenLabs Team. The sui smart contracts platform. `https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf`, 2022. Accessed on Aug 24th 2023.

[13] Abci++. `https://docs.cometbft.com/v0.37/spec/abci/`. Accessed on Aug 24th 2023.

[14] Comet bft. `https://cometbft.com`. Accessed on Aug 24th 2023.

[15] Interplanetary consensus. `https://ipc.space`.

[16] Mir - the distributed protocol implementation framework. `https://github.com/filecoin-project/mir`. Accessed on Aug 24th 2023.

[17] The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. `https://aptos.dev/aptos-white-paper/`, 2022.

[18] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. `https://github.com/ethereum/yellowpaper`, 2014.