CYBERNETICA

Integration of Sharemind MPC into Carbyne Stack

Technical Report

Version 1.1

29.12.2023

D-2-502

Public

Date	Version	Description
31.01.2023	1.0	Initial version
29.12.2023	1.1	Updated integration plan milestones

Table of Contents

1 Introduction

1.1 Purpose

For a year leading up to this report has the Carbyne Stack open source project had the attention of the Sharemind MPC team. The project's significance to us is multifaceted. First, it recognizes the challenges of MPC deployments and offers a cloud-native stack that is easy to deploy and configure. Secondly, it proposes an API which is not necessarily specific to the underlying MPC backend. Finally, its potential for scaling MPC over a cluster. These qualities justify adapting the core tools and protocols of Sharemind MPC to run inside Carbyne Stack's managed environments. This document seeks to answer the *what*, *why*, and *how* in regard to this integration.

The first part of this report expands on the publicly available documentation of Carbyne Stack based on insight from various secondary resources (source code, meetings, etc.). It does so in a comparative manner, describing how certain details compare to Sharemind MPC, with the goal of making it more comprehensive to ones who are more familiar with one of the platforms, but less so with the other. The second part outlines the motivations behind the integration. The final part explores technicalities of a possible integration, the purpose of which is to lay groundwork for the starting of development and invoke discussions.

A reader of this report is expected to have a general understanding of secure multi-party computation and core concepts of both the Carbyne Stack[1] and Sharemind MPC[2] platforms.

1.2 Scope

This report first and foremost explains the plan of integrating the two MPC platforms, from conception to a state of supporting the main features of Sharemind MPC. Additionally, some extended goals are outlined.

Details which are deemed significant for the integration are explored on a more technical level, while some questions are left open-ended. This report is subject to new revisions as the Carbyne Stack project matures over time, filling in gaps that can not be planned for at this time.

1.3 Overview

Chapter 2 introduces and compares the platforms. Chapter 3 explains the integration's motivation. The draft plan along with technical aspects and open problems is described in chapter 4.

1.4 Definitions, Acronyms and Abbreviations

VCP / computing party
Virtual Cloud Provider – a single MPC party which provides an MPC computation service

VC / clique
Virtual Cloud – a set of VCPs which jointly engage in MPC computation

Pod
smallest deployable unit in Kubernetes; usually a single container

[1]Carbyne Stack introductory video: `https://carbynestack.io/getting-started/`
[2]Overview of Sharemind MPC: `https://docs.sharemind.cyber.ee/2022.03/prologue`

(MPC) task
execution of a single MPC program

ACL
Access-control list

IaC
Infrastructure as Code

FaaS
Function-as-a-Service or serverless computing

ACID
Atomicity, Consistency, Isolation, Durability – defining properties of a database transaction

2 Comparison of the Two Platforms

Carbyne Stack is a software stack intended to fill the market void of an enterprise focused MPC platform that is open-source and community maintained [1]. It can be characterized as a set of middleware and supporting facilities for clients to invoke MPC programs and store encrypted data over multiple Carbyne Stack deployments acting as computing parties. It is noteworthy that

1. it follows a client-server model of MPC, in which clients – data providers and data recipients – do not partake in the computation, but rather offload the computation along with any encrypted inputs to the computing parties' clusters;
2. the platform does not itself supply, and is not technically constrained to any MPC backend – presently it leverages the MP-SPDZ [2] project for its protocols and runtime.

At a glance, Sharemind MPC bears resemblance to Carbyne Stack with regard to the client-server model, its general purpose and functionality. It does however incorporate its own runtime and set of MPC protocols, which are tightly coupled to the platform as a whole.

With both platforms aiming to provide a similar MPC pipeline, they share a number of architectural challenges. The following section compares the differences in solving some of these challenges based on the implementations at the time of writing.

2.1 Services

Carbyne Stack adopts a *cloud native*[1] approach which constitutes a microservices architecture. As with many other cloud native projects its services largely rely on the Kubernetes API and other technologies of the CNCF[2] ecosystem. It is split into the following services:

1. Ephemeral is a FaaS that executes MPC computation processes per clients' requests. Knative Serving[3], a solution for FaaS within Kubernetes, is used to scale Ephemeral instances down to zero when idle, and transparently spawn pods as requests arrive. This effectively distributes MPC computation throughout the cluster.
2. Network Controller lives in Kubernetes' control plane, monitoring the cluster for new MPC tasks, for which it exposes a cluster port. Istio's Virtual Services[4] are used to create and manage those routes.
3. Discovery is a stateful service that is used by Ephemeral instances to register the cluster endpoint assigned to the task and fetch the public endpoints of the neighbouring clusters' task. The Discovery service of each party communicates with its counterparts to learn the TCP ports for the tasks.
4. Amphora is the service for storing and receiving secret shared values from cloud or in-cluster object storage.
5. Castor is the storage service for correlated randomness.
6. Klyshko is responsible for running the *offline phase* of MPC to generate correlated randomness.

[1] The concept of cloud native: `https://aws.amazon.com/what-is/cloud-native/`
[2] CNCF - *Cloud Native Computing Foundation*: `https://www.cncf.io/`
[3] Knative Serving: `https://knative.dev/docs/serving/`
[4] Virtual service: `https://istio.io/latest/docs/concepts/traffic-management/#virtual-services`

Neither a microservices architecture or distribution of computation over nodes has been embraced by Sharemind MPC. Instead the central server-side component, called the application server, has a monolithic architecture. It does however host a loadable module system to add new functionality to the virtual machine running the MPC program. Some modules act as services within the application server which can be called from the MPC program's bytecode and other modules.

Similarly to Ephemeral, the application server on its own is only responsible for handling client sessions, starting virtual machines, and managing the bytecode programs. Facilities such as Carbyne Stack's Network Controller and Discovery are not necessary due to a fixed network topology that is coordinated and configured ahead of time – only one port is utilized, with messages routed to respective sessions and facilities within the application server, referred to as multiplexing.

`tabledb` and `keydb` modules provide MPC programs access to tabular and key-value databases. These modules have a database management layer which synchronizes write operations with other computing parties. Amphora's key-value nature expects keys as UUIDs supplied by the client and therefore does not require synchronization of writes.

Castor and Klyshko have no counterparts in Sharemind MPC for the time being. Future development in active security protocols may alter the fact.

2.2 Environment

Either platform has distinct requirements for its environments. With the cloud native approach, Carbyne Stack relies on Kubernetes to provide a well defined environment. It follows that the platform comprises a set of declarative configurations for Kubernetes, describing among others the storage and networking requirements for the services. A Kubernetes cluster that can provide a public IP and persistent storage should therefore be sufficient to host the deployment.

The Sharemind MPC application server and accompanying libraries are generally installed as packages onto supported Linux distributions running on bare metal or a virtual machine. Some dependencies rely on packages available to the host operating system through its official repositories, although containerization or virtualization help avoid this.

While Kubernetes clusters are often already configured for external access, the Sharemind MPC application server requires a public IP or its main TCP port to be forwarded to the host from outside the local network.

Sharemind MPC can store intermediate results in either the filesystem or a Redis datastore. The filesystem option requires the host to have a sufficient amount of reliable disk space, while the latter expects a separate Redis instance accessible from the host.

2.3 Deployment and Setup

Either platform requires involved cooperation and configuration for setup by the corresponding administrators. Mutually, they need to exchange the public IP their services are accessible from. With Sharemind MPC, its further necessary to supply all parties with each other's certificates and the certificates of clients, which is the basis of mutual authentication and TLS going forward. Computing parties are responsible for auditing and submitting code into their Sharemind MPC instance and coordinate the ACL configuration to be equal between one another. Carbyne Stack's authentication of clients and neighbouring parties, as well as code auditing is yet a work in progress.

Even with a microservices architecture Carbyne Stack offers a relatively simple overall deployment with the Helm package manager, with cloud specific IaC in the works to improve the process further.

2.4 Client Interface

Every client accessible service in Carbyne Stack is exposed as a HTTP, with Java client libraries and CLI programs provided. Sharemind MPC utilizes a proprietary protocol over TCP for client communication – the native C++ API, CLI programs, and HTTP gateway libraries are among the available options for clients to communicate with the VC.

2.4.1 I/O

All input and output in Sharemind MPC is handled via arguments and published values[5] of tasks. By that, clients have no direct access to the persistence facilities. Carbyne Stack operates in the opposite manner – clients can only use data available to the Amphora storage in a task, with task results subsequently stored in the database. If clients upload values to Amphora or their task finishes with a result, they are left with a key corresponding to that value, which can be used to fetch the value or be used as input to a new task.

Amphora's current implementation is specific to the SPDZ protocol, that requires message authentication codes (MAC) accompany each secret share for active security. Its API facilitates the verification of sharings on fetching secrets from the VC and creation of authenticated shares on upload. Sharemind MPC does not have facilities for operations on authenticated secret shares.

As of now, the I/O possibilities of Carbyne Stack are restrictive

1. Amphora only supports secret shared integer vectors;
2. Ephemeral can only write a single output share vector to Amphora per task;
3. MP-SPDZ tasks have no access to data other than arguments (vectors specified at task activation) specified by the client, making the processing of larger datasets cumbersome.

Tasks in Sharemind MPC can read and write to tabular (HDF5) or Redis key-value databases during execution. Integers, floating point numbers, booleans, vectors and strings are supported throughout.

2.4.2 Session Management

In both cases, the client's task initiation triggers a long-running request, either in the form of a persistent TCP connection for Sharemind MPC or HTTP for Carbyne stack, for the duration of the task. Subsequently, the response holds the result of the computation.

2.4.3 Access and Policy Control

Sharemind MPC's client authorization works on both database (table/key, read/write) and program level, specifying the programs each client is allowed to execute. This enables specification of intricate MPC applications with clients of various roles.

[5] private values revealed to the client

3 Motivation

There are several justifications to undertake an integration of the Sharemind MPC runtime into Carbyne Stack.

Distributing the effort and knowhow of engineering the complex distributed system, which can lead to a more transparent, well thought out, and standard way of realizing client-server MPC. Although MPC is distributed computing *per se*, it is often incompatible with foundational techniques of distributed systems e.g. computing parties having unproportional influence over the system due to the coordinator/master-slave pattern. Carbyne Stack, being an open source and community driven project, might help to reimagine and reimplement the measures we have so far invented for the application server with better qualities (reliability and resiliency).

Contributing to Carbyne Stack's aspiration for becoming an **industry standard for MPC** can be mutually beneficial: Sharemind MPC could adopt the unified API early on in the process to advance interoperability while refining Carbyne Stack's software requirements.

Parallel task execution and scaling comes naturally from how Carbyne Stack utilizes the nodes in a Kubernetes cluster to distribute MPC computation. It's observed, that task-level parallelization of MPC applications is a surefire way to improve execution times [3]. Unfortunately the current architecture of Sharemind MPC can not benefit from this in full potential, as concurrent tasks compete for network and compute resources. Both of these issues are helped by logical and physical distribution of tasks within a VCP's cluster.

Augment Carbyne Stack with a range of Sharemind MPC features like the Protocol DSL for protocol development and the SecreC language with its Analytics Library and standard library for writing optimized MPC programs. Further, this will enable previously developed Sharemind MPC applications to fully utilize modern environments such as commodity clouds.

Seamless cloud deployment procedures provide computing parties with tools and documentation to deploy a reproducible environment in commodity clouds. Carbyne Stack has a head start in this regard, with most cloud providers offering managed Kubernetes control planes, and cloud-specific IaC scripts are in the works. This has potential to reduce the assistance of set up needed from the technology provider to a minimum – something that hampers on-premises and bare metal deployments.

4 Integration Plan

4.1 Goals

The bottom line of the integration is to engineer a minimal composition of Sharemind MPC components into a fully functional MPC runtime that is operated in a similar fashion to the current MP-SPDZ binaries. The following are expectations for both the integration and the resulting architecture changes of Sharemind MPC.

1. The Sharemind MPC runtime should be interchangable with the current MP-SPDZ variant. The backend choice should be a matter of a configuration change.
2. The new runtime comes in the form of an executable, spawnable by Ephemeral as it currently happens with MP-SPDZ.
3. The executable would contain the minimum to run a single MPC task (henceforth referred to as *task-as-process*) but retain the characteristic feature-set of Sharemind MPC, including SecreC, protocols and protocol DSL, previously written algorithms, etc.
4. The resulting architectural changes should not irrecoverably branch from the Sharemind MPC product. The task-as-process track is to eventually be a part of standalone Sharemind MPC along with any necessary accompanying services.

4.2 High Level Vision

The envisioned runtime is essentially a trimmed-down version of the Sharemind MPC application server. The trimming naturally begins from the functionality that Carbyne Stack already solves with its services (see section 2.1) or is expected to support in the future. The list of such responsibilities that are therefore transfered to Ephemeral and other services are as follows:

1. listening for clients' task invocations and executing the task;
2. task lifecycle management;
3. coordinating with neighbouring VCPs;
4. handling of task arguments and published values;
5. client authentication*;
6. client authorization, access and policy control*;
7. securing protocol communication between task instances**;
8. submission and storage of MPC programs**;
9. guarantee transactionality of tabular secret storage facilities**;
10. session management for a scheduled execution model**.

Asterisk (*) marks the feature which is currently worked on; double asterisks (**) mark the features which do not yet have a proposal.

This would leave the runtime comprising two facilities:

1. Networking layer for protocol communication;
2. Virtual machine with the loadable module system.

4.3 Development Roadmap

Being in its early stages of development, many technical challenges still lie ahead for Carbyne Stack itself before being ready for production use. A few examples of this were already listed in section 4.2 (items 5, 6, 7, 8) – elements not exclusive to this integration but rather necessities for client-server MPC – are subject to being realized through enhancement proposals. We can't predict the outcome of these developments (although we plan on participating in these processes), so we don't focus on these aspects in this section.

The integration plan is presented as a roadmap of milestones, that are adjusted and added as Carbyne Stack and the integration itself matures. Further, this gives a better overview on what functionality we sacrifice and regain over the integration period.

Milestones with lesser impact or such that can be considered enhancements are listed in no particular order in appendix A.

4.3.1 Milestone 1: A Minimal Runtime

This milestone's development focus is on the Sharemind MPC side, with the goal of creating the Sharemind MPC task-as-process runtime. Its starting point is the application server; the end marks a PoC of a SecreC task that can be run by Ephemeral with only small changes.

The transition from the application server to a single process per task removes the need for much of the complexity which was specific to the service platform – isolation of concurrent sessions and stream multiplexing to list a few. It subsequently does not make effort to preserve most of the fundamental functionality of Sharemind MPC. This includes the distributed databases and task I/O, client controller along with all access control features, consensus streams, and multiple concurrent protection domains[1]. Hence during this milestone, supporting only such programs which take no arguments and publish no results is sufficient, even though it provides no realistic utility.

We assume the following capabilities from the runtime:

1. initialize the virtual machine and protection domain(s);
2. establish connections to counterpart runtimes during initialization
   a. without mutual authentication and transport layer security;
   b. skipping client process negotiation and validation of bytecode checksum;
   c. an established connection is further only used by the protection domain;
3. independently run computation until finished.

Item 2c allows for major simplifications to the networking layer if MPC programs were to only support one protection domain at a time. Otherwise, multiplexing can not be eliminated.

The executable would require at least the following arguments:

1. IP and port of each neighbouring party's task instance;
2. MPC program's bytecode.

Finally, a fork of the Ephemeral server should be created, implementing the generic `MPCEngine` interface that executes the new runtime. The set-up is illustrated on figures 1 and 2.

[1]Multiple protection domains refers to the feature of Sharemind MPC to be configured with several VC topologies, that can utilize different protocols within a single SecreC program.

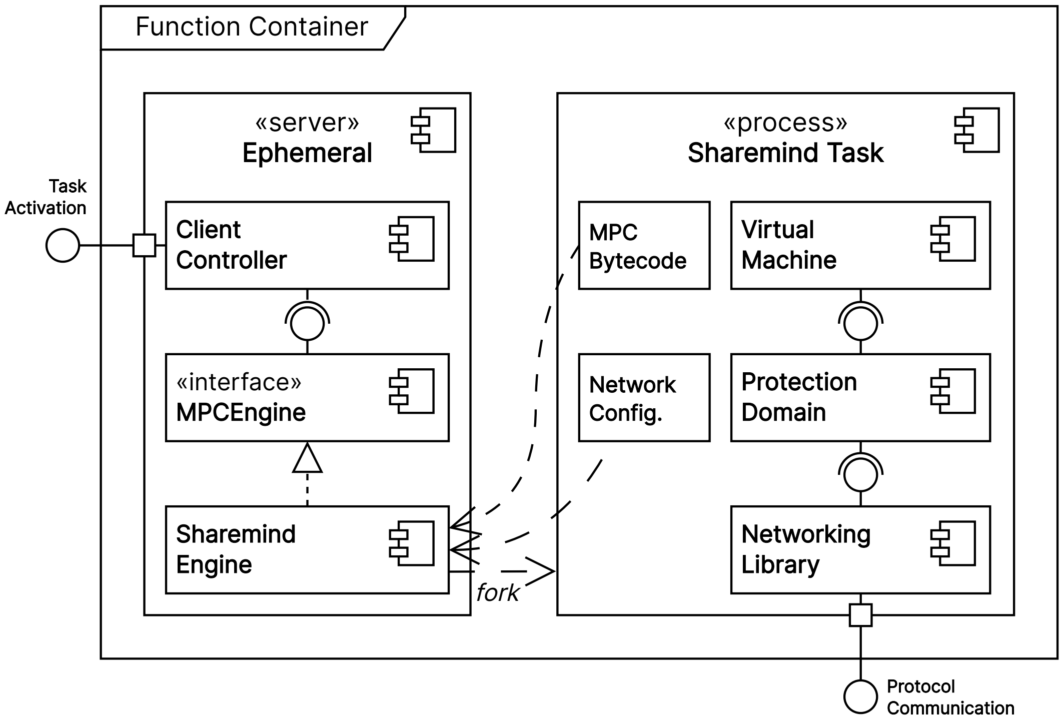


Figure 1. Component diagram showing the relation of the new runtime and Ephemeral

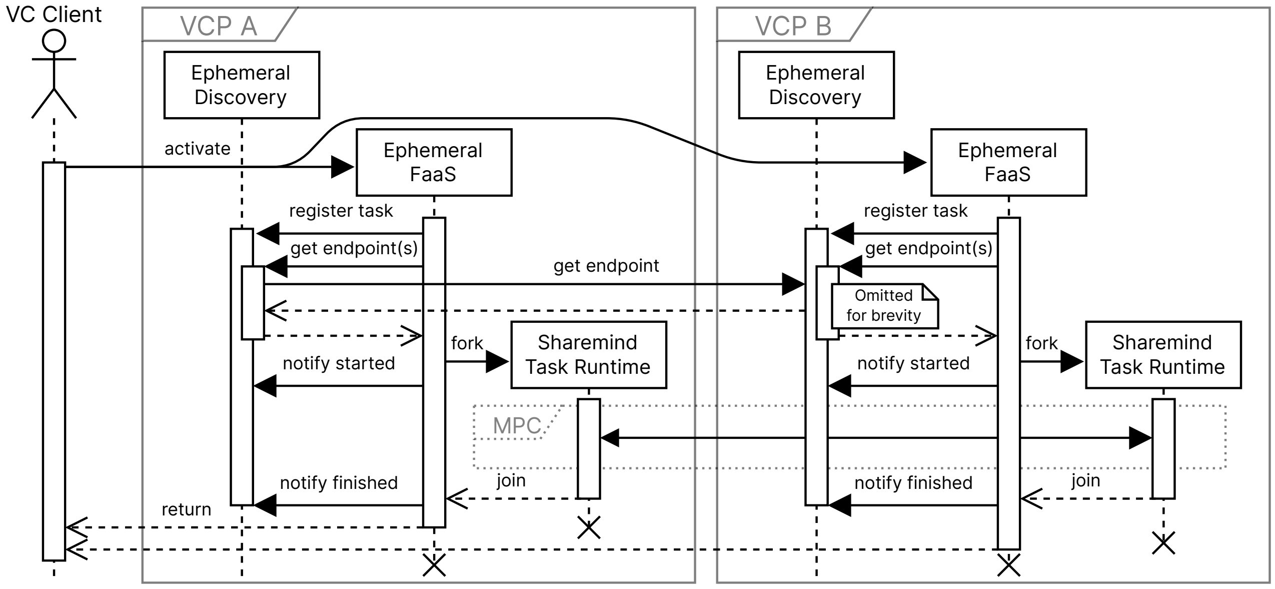


Figure 2. Sequence diagram illustrating the order of operations for invoking an MPC task within a two-party virtual cloud.

4.3.2 Milestone 2: Task Arguments and Published Values

This milestone targets both Carbyne Stack and the runtime to implement client supplied arguments and returned values.

SecreC programs rely on named arguments as inputs, typically transmitted from the client to the server as *binary argument streams* alongside the task execution command. The streams comprise secret shared and public values; there's a separate stream for each VCP. In this iteration, in accordance with Carbyne Stack's distinct procedure for task I/O (described in section 2.4.1), clients upload the argument streams as a single object to Amphora, later referencing it in their task activation. The argument streams will still be constructed by the client at the time of secret sharing.

Socket communication is used between Ephemeral and MP-SPDZ, however in that case the stream consists of an ordered sequence of secrets (i.e. multiple Amphora objects), rather than named arguments – Ephemeral feeds values in the same order that the business logic expects them. As SecreC imposes that arguments can be loaded into the program in any order or ignored altogether, the runtime would either have to query the parent process for individual arguments or ingest the whole available argument stream at initialization to make it available to the virtual machine during execution. This milestone will implement the latter for simplicity, but further iterations may strive to imitate the existing I/O convention further[2].

The Ephemeral instance spawning a runtime feeds the referenced argument stream, fetched from Amphora, in whole to the runtime, reusing the socket interface. Published values are marshalled using the same representation, communicated back to Ephemeral and stored in Amphora. With this, the handling of inputs and outputs remains largely unchanged between the two runtimes.

The work that needs to carried out on is as follows:

1. extend the Amphora client to accept binary *argument stream* objects as uploaded values, bypassing the *input supply* protocol as this is only needed for authenticated shares;
2. create an Amphora service endpoint to accept aforementioned objects;
3. reintroduce the Amphora feeder from the existing `SPDZEngine` implementation in the `MPCEngine` implementation created in milestone 1 to provide the *arguments object* to the runtime via socket; take similar action for published values;
4. have the runtime
   a. read, parse, and cache arguments, making them available for the virtual machine;
   b. collect, marshal, and send published values back to Ephemeral.

4.3.3 Milestone 3: Distributed Tabular Databases

Sharemind MPC owes much of its performance to vectorized operations on secret data to batch together protocol messages. Specific language constructs in SecreC enable writing code in a SIMD manner, which is highly encouraged over processing scalars and using loops. This and the utility factor of structured data in analytical applications has influenced the persistent storage method of choice in Sharemind MPC to be columnar databases with its `tabledb` module. Numerous existing applications, Rmind among others, depend on its API. With the interest of ensuring backwards compability, accessing tabular data within MPC programs is required.

[2] Two-way inter-process communication between Ephemeral and the runtime is expected to facilitate argument querying. Milestone 3 addresses a similar pattern for data tables in a way that would be transferrable to this as well.

The `tabledb` API allows SecreC programs to read columns, create, delete or append to database tables with names defined at execution time.

Application server's current database engine relies on files in the HDF5 file format to store the data. An issue with this approach in a Kubernetes cluster is the absence of a common filesystem, unless Persistent Volumes with a ReadWriteMany access mode are used. This could be possible using a cloud file storage solution like Amazon's EFS, which is non-portable, or in-cluster file storage solutions like Rook Ceph, which is heavyweight and stores data in temporary node storage [3].

Utilizing an existing DBMS might seem natural, however the benefits of these in our scenario are limited. For one, due to the simplicity of `tabledb` operations there is no need for powerful query or data manipulation capabilities. Moreover, maintaining a consistent state of secret shared data tables can not be achieved solely with any traditional distributed database offerings – the consistency requirement spans over the VC, but each VCP holds unique shares for values. Access pattern is also non-traditional as each individual VCP simultaneously queries data expecting the same values as its neighbours. Considering there may be many tasks operating on the same data at once, an extra concurrency control layer is unavoidable. For example, the application server applies a consensus protocol to ensure correct order of writes [4].

The proposed alternative is to build a table abstraction on top of a cloud storage medium alongside a dedicated service implementing the `tabledb` API. Tables' contents would be kept in object storage, while metadata is tracked in the etcd key-value store. The components' relations can be seen on figure 3. Being append-only, there is no drawback to storing segments of tables as immutable objects; tables are growable by creating subsequent segment objects and having the relevant pointer stored in etcd. The database service, local to each VCP, interfaces with other local services to fulfil requests issued by the runtime.

Etcd and the service in conjunction enact consistency and concurrency control for database operations. The lease mechanism, linearizability, and *multiversion concurrency control* (MVCC) make etcd a prime tool for building purpose specific distributed coordination mechanisms[3]. Put simply, each VCP provisions an etcd node forming a VC-wide etcd cluster. Each type of database operation defines a procedure between the service and its co-located etcd node – it may include lock acquisition, commit decisions, and other nuances that call for consensus. Etcd's replication is the only necessary means of inter-VCP communication in this regard, within itself it ensures strong ACID compliance which is exploited in our procedures to provide much needed isolation and consistency guarantees to the secret share database.

The service in question would hopefully become an extension of Amphora as it is akin in technical and practical aspects. Being the canonical persistence facility of Carbyne Stack, its hard to legitimize another similar but MPC back-end specific core service, especially as there is potential for code reuse. Both expose a CRUD[4]-like API, store secrets in object storage, and are regarded as policy enforcement points for data access. Further, while the described functionality is described to serve Ephemeral, the table interface could also be exposed to clients, eliminating the need for a dedicated MPC program managing table import and export.

Beside carrying out the aforementioned architecture, some adjustments to existing components are expected for this milestone. The runtime would not be communicating with the database service directly but via its parent process, isolating the runtime from responsibilities like keeping

[3]This usage pattern is encouraged by etcd developers `https://etcd.io/docs/v3.5/learning/why/#using-etcd-for-distributed-coordination`

[4]*Create Read Update Delete*

track of its game ID, client ID, and handling errors. Using the socket interface, it should communicate the running process's requests together with data to Ephemeral, that in turn reaches out to Amphora for servicing.

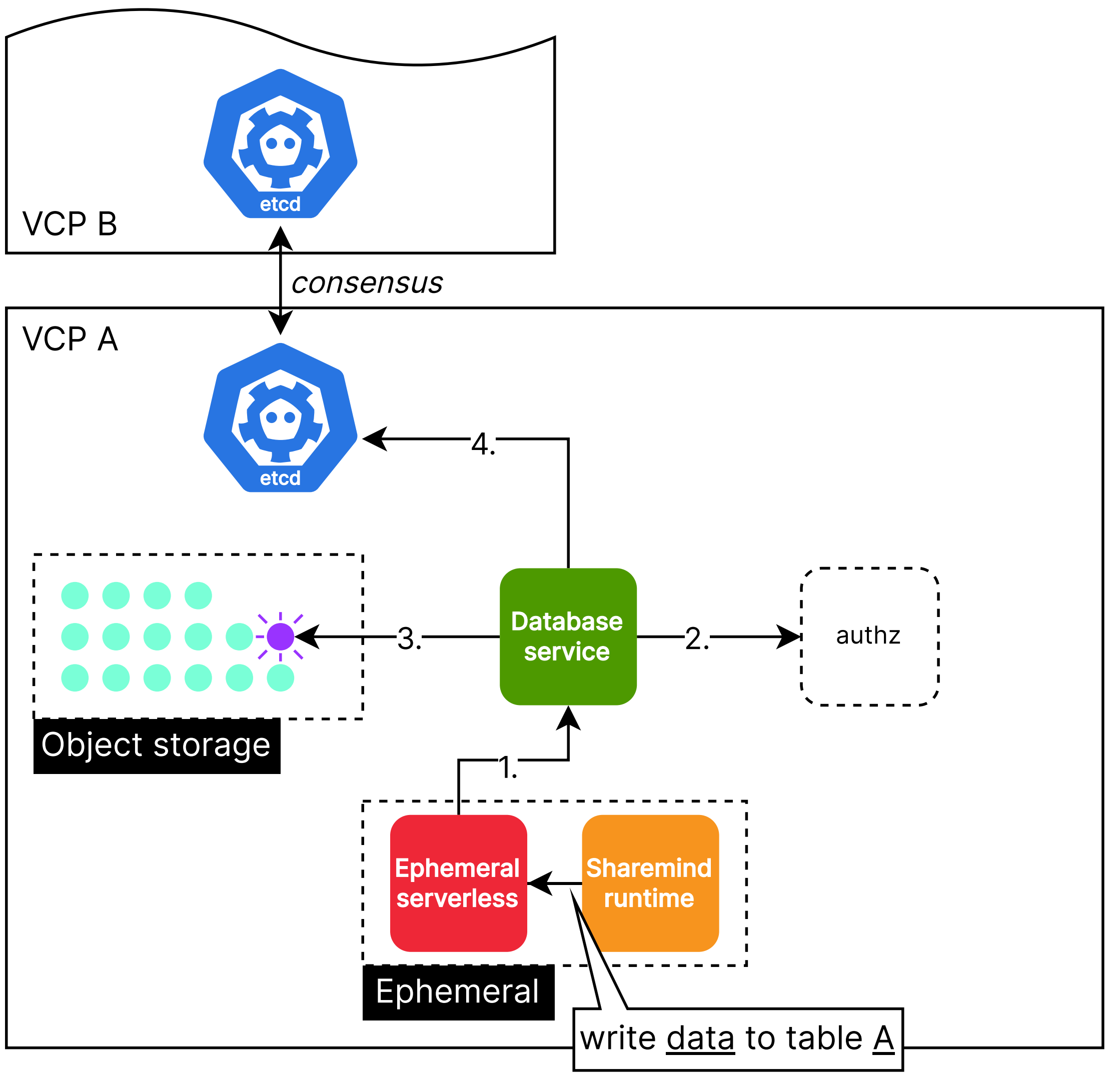


Figure 3. Flow of a write request issued by the Sharemind Runtime. The runtime orders Ephemeral to write a table segment to storage, which passes the request and data over to the local database service at (1.); the service issues an authorization check to the policy decision point at (2.); if succeeded, the data is uploaded to object storage at (3.) and a reference to the created object is submitted to the etcd cluster at (4.) in accordance with a *write procedure*, making the change durable.

Bibliography

[1] The Carbyne Stack Authors. *Carbyne Stack*. https://carbynestack.io, Last accessed on 2023-12-29. 2022.

[2] Marcel Keller. “MP-SPDZ: A Versatile Framework for Multi-Party Computation”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417872. URL: https://doi.org/10.1145/3372297.3417872.

[3] Kert Tali. “Parallel and Cloud-Native Secure Multi-Party Computation”. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=74954. MA thesis. Institute of Computer Science, University of Tartu, 2022.

[4] Cybernetica AS. *Deliverable D3.2: A protocol suite for robust distributed computations*. SEVILLA Technical document. 2016.

Appendix A Extra Milestones

A.1 Task Scheduling

Client-server MPC is often prone to failures due to intermittent network faults between the client and the server if the client has no way to resume a session. This issue is especially apparent with long-running MPC tasks that process large data volumes. We find that a task scheduling API, the likes of Hadoop Yarn or other big data processing platforms, is more fitting to client-server MPC than the current synchronous model used by both Carbyne Stack and Sharemind MPC.

A.2 Transactional Distributed Databases

As described in section 2.1 and milestone 3, tasks in Sharemind MPC have write access to databases during execution. Databases are pseudo-replicated in a manner in which every VCPs database is a mirror image of others regarding schema and shape, with each private data value representing its secret share. This evidently presents the requirement for replicas to maintain a strongly consistent state – all parties *not* reading a share corresponding to the same value or recieving an error would constitute a wrong result or failure. With Sharemind MPC the correct ordering of database operations is guaranteed due to the consensus mechanism. Completion of milestone 3 brings the same assurance on the `tabledb` API level to Carbyne Stack as well.

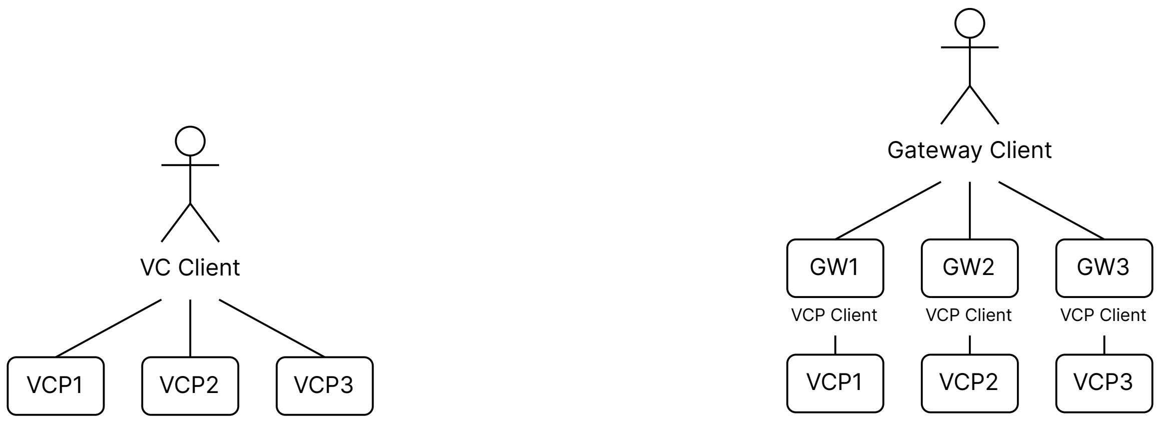
This can be argued to not be sufficient for the reliability of the MPC system. With consistency measures between the share databases, we can protect against intermittent faults and races on the database operation level. Clients of the platform are not concerned with this granularity however if the smallest work unit they can influence is a *task* that potentially composes multiple modifying operations in sequence. If a task were to fail between two database operations, the first would still have been made durable – a potential invariant violation w.r.t the task, which the client has no means of solving[1]. Based on experience, such cases call for manual intervention by VCP administrators, which is laborous enough to coordinate in a one-shot MPC set-up and would be unfathomable in any kind of MPC-as-a-Service scenario.

It's worth noting that either problem is specific and inseparable from concurrently accessed structured databases and do not apply to the current state of Carbyne Stack. The ordering issue is effectively avoided by relying on client-specified UUID based keys in Amphora. Values are loaded from the datastore at the start of the task and only persisted after its succesful completion, avoiding possibility of a partial state.

With this milestone we suggest transactional mechanisms ensuring that changes to the database are atomic on the task level. This would enable full transparency for developers in terms of mid-task failures and compability with legacy SecreC code. The architecture outlined in milestone 3 could in fact be augmented in various means to achieve task-based transaction management. For example, starting a task could acquire a long-lived lease in etcd that is attached to any un-commited changes throughout the lifetime of the task. Finishing a task would trigger a commit based on consensus, supported by etcd, that would make the changes durable.

While the requirements for these measures are not yet fully clear, the specific isolation level

[1]MPC programs could be written accounting for failure recovery procedures, however a more general solution is desirable in interest of conciseness.



(a) VC client **(b) VCP client**

Figure 4. Client models for interacting with the MPC platform.

remains an open question. Some tasks may want to exploit lower isolation levels, like multi-task parallel programs, for efficiency. Otherwise, since the `tabledb` API we focus on is constrained enough to not warrant updates, serializable isolation can be achieved by just fixing an etcd revision at the start of the task for all following table reads without bearing any additional cost. It might be that this question should be solved by an optional isolation level parameter adorned to the MPC program.

A.3 Gateway Applications

A distinction can be made between cases where the clients are data owners or data recievers who directly trigger MPC tasks, or an application initiating on its own behalf, possibly containing some public business logic. In Sharemind MPC, these applications are called (web) gateways, built on a supplied gateway library. The benefit of these gateways has so far mostly been the opportunity to integrate MPC subroutines into web applications. In this case the gateway translates HTTP from the web client to the binary over raw TCP protocol used by the application server, that can't be acommodated by a web browser. While Carbyne Stack has a native HTTP client API, support for extra server-side business logic can still be beneficial.

For clarity let's call the classic model of client, that communicates and synchronizes operations with all parties, a VC client (figure 4a), and a client that only communicates to a specific party a VCP client (figure 4b). Each gateway (GW) acting as the VCP client is usually in the same administrative domain with its VCP.

This type of deployment offers flexibility in realizing various business cases. For example, each server could buffer input and/or result shares, schedule the execution to a certain time (epoch), pipeline several tasks, coordinate parallel tasks. The VCP client method enables MPC powered applications which ingest secret shares over any medium, like streams or web clients, utilizing any authentication, without the need for a central service and therefore avoid encrypting shares for VCPs.[2]

A motivating example is the Sharemind Secure Survey, in which subjects receive a token to answer a set of questions on a distributed web application. Upon submission, the answers are secret shared in the browser among three gateways, which immediately triggers an MPC task

[2]An alternative approach is to use a central, or single gateway with a VC client to mediate shares encrypted for each VCP with a respective public key. This requires an intricate key distribution to be in place. Further the central service has to be trusted to not selectively omit inputs from the computation.

to validate that multiple choice questions are within the allowed range. The gateways therefore play a role in enforcing application specific rules in effort to avoid malicious clients poisoning the dataset. As the survey closes, the three gateways trigger the second MPC task to perform analysis and store the aggregate results for the survey owner to pull and recombine on demand.