Flowstate: A language for fault tolerant decentralized computation

Priyanka Mondal*, Owen Arden †
University of California, Santa Cruz
Email: *pmondal@ucsc.edu, †owen@soe.ucsc.edu

Introduction

Building decentralized distributed systems is very difficult because of problems like host failures, network partitions, malicious hosts etc. **C**onfidentiality, **I**ntegrity and **A**vailability policies can be used to make a distributed system secure. But enforcing these policies are hard as the participants have mutual distrust and absence of an universally trusted party makes it even harder. Some of these problems are already solved in isolation, but bringing those techniques together is a complicated task. Our goal is to build a general purpose programming model which enforces C,I,A policies in a decentralized, distributed system and also make the computation fault tolerant. We call our programming model Flowstate, where participating nodes, individually execute translated versions of an expression atomically and reach consensus on the result based on a parameter specified by the programmer. Flowstate is designed based on distributed shared memory and optimistic concurrency control model and it is compatible with any quorum based replication protocol. One of the novel approaches in this work is this model supports secure information flow to and from multiple clusters at the same time. The computation nodes, which we call **worker** nodes, and the memory locations at replicas, which we call **stores**, are annotated with integrity, confidentiality and availability labels to enforce secure information flow and guarantee availability. Each worker maintains a log during the computation. A sufficient number of these logs are required as a proof of combined authority from the workers to commit the changes at the stores. The expression replicated at the workers is confidential i.e it is **encrypted**. The workers are able to execute the expression only if they have the authority to read it. After the computation each worker adds its **digital signature** to the end result and the logs as a proof that those are computed by them. In our work computational availability is different from data availability. The availability of the end results depend both on computational and data availability. Computational availability is ensured by **replicating** the computation in to multiple workers, while data availability is ensured by replicating data in to a quorum of stores.

Contributions

System Design There can be three kinds of nodes in our system. A **Client** node requests a set of workers to compute an expression e and return the answer within a certain amount of time. There can be many clients in the system but every client communicates with the workers independent of each other. **Worker** nodes are the execution nodes. Reads and writes done by the workers happen inside **transactions** and are recorded in their logs. When an execution ends, the worker logs should match, although the integrity label may differ. These logs are combined together to commit values to the storage nodes. The workers choose a leader amongst themselves who is responsible for coordination among the workers and communicating with the stores. Each worker has its local cache. **Storage** nodes or stores are distributed into clusters. Each cluster replicates a memory locations into a quorum of nodes. The nodes in a cluster choose a leader amongst themselves. A worker requests a store for a memory location when a cache miss happens. These requests are always forwarded to the leader in that cluster. Each memory location stores a $< value, version >$ pair. The $version$ is incremented by one when the value is updated. We use **shared memory** model in our language design. The cache memories at workers and the memories at the storage nodes together build the shared memory. The execution at the workers follows **optimistic concurrency control** model. The workers assume their caches are up to dated and they start computation, of an expression e, right away. There can be two instances where Flowstate might update the worker caches and resume the execution of e from the beginning. The best scenario would be the following two cases never happen.

1) When Flowstate system finds that the workers' results diverge from each other to such an extent that consensus on logs can never happen, the worker caches are aggregated and the execution of e starts from the beginning.
2) Committing the end results to the stores fail when the workers reached consensus on the logs but the computation was done on old data versions. Thus the worker caches are updated with the latest values from the stores and the execution of e starts again from the beginning.

The Flowstate Language Flowstate has two kinds of operational semantics: global and local. A Flowstate program starts from a client host and then gets replicated to the worker nodes. The distributed execution which happens from the client's perspective is captured in the global semantics. The execution which happens at each individual worker node is captured in the local semantics. An expression that the client initiates executing looks like: `sync` $e@W$. Here e is the expression that the worker nodes should execute individually, and W is the consensus parameter. This W specifies how much fault the

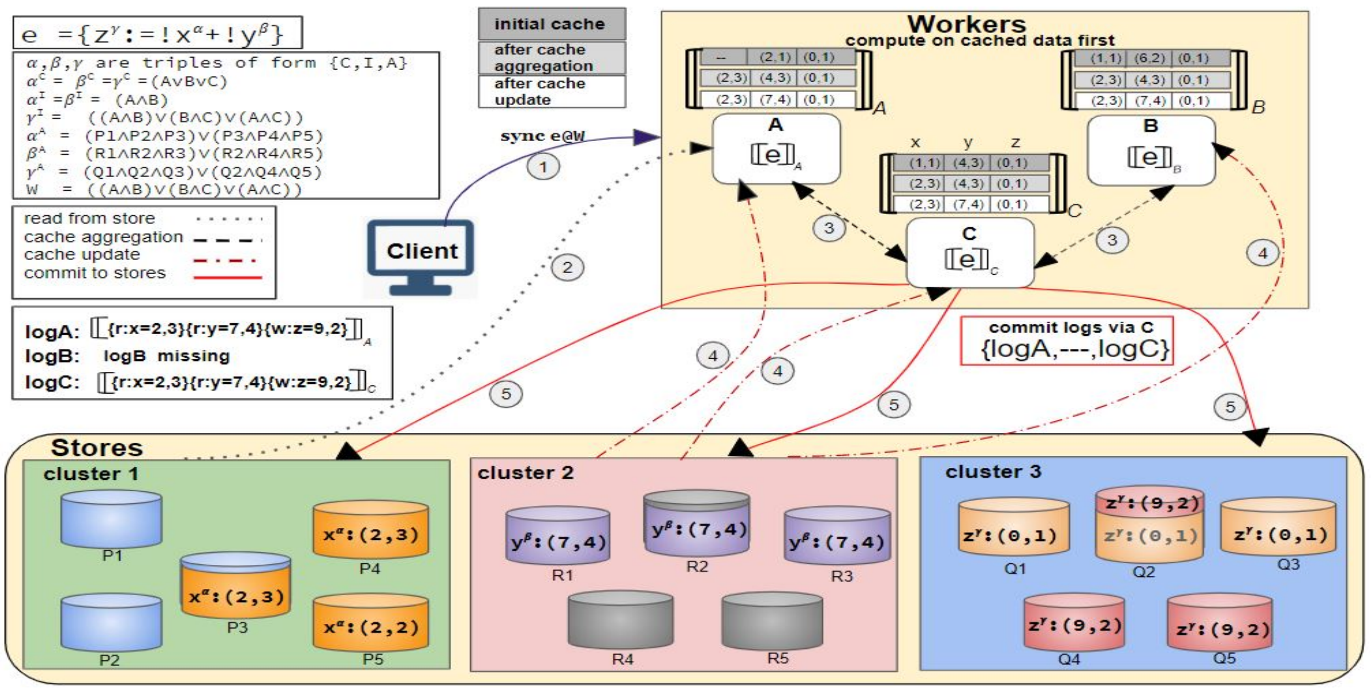

Fig. 1: Steps while executing a **Flowstate** expression e in a distributed, decentralized environment.

system should tolerate, or in other words how much computational availability is required for the execution to pass. In the example in fig.1 we have $W = (A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$, that means any two of A, B, and C need to compute same results. Thus if one of the nodes is unavailable or computes wrong result ,the overall computation does not fail. The input program e and the consensus parameter W are the inputs from the programmer. The expression that the *client* wants to execute might have integrity more than any worker in expression W. So the workers execute a translated version of e that is attenuated to their integrity label. We use a translation function $[\![\;]\!]_w$ to translates the expression's integrity while executing it at w. To maintain consistency of integrity of expressions through out the execution we extended this translation over memory, logs, and types. A memory location of type τ when stored at worker w's cache, its type becomes $[\![\tau]\!]_w$, i.e. $[\![m^\tau]\!]_w = m^{[\![\tau]\!]_w}$. When sync $e@W$ is executed at *client*, expression $[\![e]\!]_A$ is executed at A, $[\![e]\!]_B$ is executed at B and $[\![e]\!]_C$ is executed at C. At the end all the workers' logs are combined to commit the changes at the stores using a replication protocol. The Flowstate type system checks if the program has any information flow violation during compile time. We prove that results computed by a Flowstate program in a distributed environment is same as the results computed by the same program in a trusted third party.

Example Fig.1 shows different steps of executing an expression sync $e@W$. Different kinds of arrows represent different kinds of communications between the nodes. C is the leader among the workers for this particular instance. Expression e refers to three different memory locations x,y and z from three different clusters. x,y and z have same confidentiality, $(A \vee B \vee C)$. This means x,y and z can be read by policies with confidentiality $\geq (A \vee B \vee) C$. x and y have same integrity, $(A \wedge B)$. z has integrity $((A \wedge B) \vee (B \wedge C) \vee (A \wedge C))$, which means any two of A,B and C's digital signature is enough to update z. Data availability of x, y and z are different as they are stored in different cluster(shown in fig.1). The workers read from x and y and writes to z. They need to read z also to know its $version$ number (although the logs do not reflect it because it is implicit that when you do a write you do a read of that location). Initially z was stored at stores $Q1, Q2$ and $Q3$ in cluster 3. The arrows are marked with sequence numbers to denote the order of the steps. (1) The *client* replicates execution of e at A, B and C (nodes are named after the policies they support). (2) Cache miss happens at A for x, thus a read from cluster 1 takes place. (3) The caches of the workers do not agree, so cache aggregation happens. (4) The workers fail to commit as they computed e on an older version of y, so cache updates happen. (5) Workers again compute e on updated cache. C being the leader sends the combined logs to the clusters to commit the results. Once the commit is successful the caches and the clusters are updated with z's new value. The end results are not reflected in the caches due to lack of space. In step 5 commit messages are sent to all the three clusters. It is shown that B's log is missing. But that does not affect the commit, as logs from A and C are sufficient to proof enough integrity $((A \wedge C) \geq \gamma^I = ((A \wedge B) \vee (B \wedge C) \vee (A \wedge C)))$ to update z at cluster 3(in stores $Q4, Q5$ and $Q2$). One thing to notice is that the availability of the execution depends on availability of W and availability of the memory locations x, y and z at their respective clusters.