Applications of weakly consistent distributed data structures over shared logs

Pranav Maddi

Contents

1 Background 1

1.1 Shared logs 1

1.2 Consistency 1

2 Software Defined Networking 2

2.1 ONOS 2

2.2 OpenDaylight 3

2.3 Comparison of SDN systems 4

3 Common Data Structures 4

4 Proposed SDN extensions 4

4.1 Georeplication 4

4.2 Coordination 4

4.3 ONOS 5

4.4 OpenDaylight 5

5 Conclusion 5

Abstract

1 Background

I chose these SDN systems because they are open source and well documented

1.1 Shared logs

A set of totally ordered shared logs can provide weaker consistency on the data structure built on top of it. Each element in the logs have a set of dependencies. That is, to know element `e`, all of the elements of `e.dependencies` must be loaded. By allowing dependencies to be specified between elements of various shared logs, arbitrary partial orderings of nodes can be constructed while preserving the convenience of having totally ordered logs. By carefully selecting what operations on the higher level data structure require global reads and writes, causal ordering can be guaranteed across the network.

Tango [1] aims to provide a way to store metadata by allowing transactional access to arbitrary data structures with persistence and high availability. The Tango paper notes that some services such as ZooKeeper provide persistence, high availability, and strong consistency—but only for specific data structures.

Tango objects are a class of data structures stored in memory that built on the shared log. The history of the object is stored directly in the log, and each object can have multiple views which are copies of the data structure stored by the clients. The source of truth in the system is the shared log, and the views are "soft state" that can be reconstructed by replaying the shared history. Modifications to Tango objects are made by appending updates to the history, and accesses are made by syncing the view with the history.

Using a shared log makes strongly consistent operations easier than they would be in a conventional distributed system. Such applications include remote mirroring, coordinated rollbacks, and consistent snapshots across objects. These options are available since views can be synced to any offset in the log and replayed as needed.

The shared log abstraction represents a total ordering of the updates added to the log. This abstraction is a common way to organize changes to a file system, for instance [5]. By definition, this means that it is a set with a relation that satisfies reflexivity, transitivity and symmetry, where any two items can be compared. On the other hand, we want to represent a partial ordering of the updates so that we can allow the user the flexibility of not relying on other users writes. A directed acyclic graph is a data structure that can represent partial ordering relations. Initially, we considered using a shared DAG as a representation of state in the system. However, this seemed to leave the system too unconstrained since it would be difficult to maintain ordering across nodes and read forward to get new data.

1.2 Consistency

Not all applications require a strong consistency guarantee. Depending on the particular application, lower latency reads or the ability to write immediately could be preferred. We would like to enable our users to select what level of consistency works for them. Rather than have the consistency level be specified at development time, we would like to support this at the API level. By having something like a Service Level Agreement specified within the update_helper and query_helper, the client could configure the guarantee that works best in the situation. The SLA specified would then inform where in the DAG to place the update. The motivation for such a system is that different procedures in an application can have different guarantee requirements.

SLA requirements we may support include: [6]

strong returns the value of the last preceding Put(key) performed by any client

read-my-writes returns the value written by the last preceding Put(key) in the same session or returns a later version; if no Puts have been performed to this key in this session, then the Get may return any previous value as in eventual consistency.

bounded(t) returns a value that is stale by at most t seconds. Specifically, it returns the value of the latest Put(key) that completed more than t seconds ago or some more recent version.

causal returns the value of a latest Put(key) that causally precedes it or returns some later version. The causal precedence relation $<$ is defined such that op1 $<$ op2 if either (a) op1 occurs before op2 in the same session, (b) op1 is a Put(key) and op2 is a Get(key) that returns the version put in op1, or (c) for some op3, op1 $<$ op3 and op3 $<$ op2

Any of these consistencies could be supported with the multiple shared log structure, as long we carefully implement the distributed data structure on top of it.

2 Software Defined Networking

In general, SDN allows central control of a network. It aims to keep the complexity of the network manageable by providing abstractions for the underlying network devices. SDN provides four main advantages over traditional networks:

- Separation of the control and data planes
- Centralization of the control plane
- Programmability of the control plane
- Standardization of API's

[3]

Networking protocols are often arranged into data and control planes. The data plane consists of all the messages that are generated by the network clients. The network must then decide how to transport the packets from source to destination, which requires some level of decision making by the control layer. In a traditional network, these decisions are made by the control layer of the router which determines which routing behavior to perform. This might include computing L3 or L2 routing protocols such as Open Shortest Path First or Spanning Tree. A network manager might keep track of traffic statistics and the network state.

With SDN, this control logic is both separate and centralized, away from the data plane. The data plane still forwards the packets using the forwarding table assigned by the control plane, but that logic is separated out into a controller. This has the added benefit of reducing the cost of the switches. SDN's centralization of control makes determining the network state faster than with standard distributed network practices, and policy changes can be propagated much faster. Centralized control in SDN networks also have scaling issues, and it is necessary to divide the network into subnets that can share control strategy.

OpenFlow

OpenFlow is the protocol for communicating with the routers and switches. It allows the controller to install flows in the system.

2.1 ONOS

[2] ONOS is an open source network "operating system" that aims to have distributed control of the network. Like the other systems covered here, ONOS uses the OpenFlow control-data plane interface. The overall architecture of ONOS is composed of a network view that contains a network graph abstraction, OpenFlow managers, as well as a Zookeeper registry for coordination.

To interact with the OpenFlow interface, ONOS uses modules from Floodlight, which is another SDN implementation. This includes the switch manager, the IO loop, and link discovery. These modules as well as shared Floodlight module manager means that ONOS is slightly more compatible with other SDN controllers.

The network view data model was stored in an ONOS graph database built on top of a distributed and persistent key-value store called *RamCloud*. RamCloud exposes the Blueprint graph API which applications use to access the network state. The system guarantees eventual consistency in the graph view.

The transition semantics of the graph database are maintained to correspond with structural nature of the graph, such as requiring that an edge connects two nodes. To justify that eventual consistency is good enough for the graph view, some of the ONOS experiments show that when using redundant and identical flow path computation on all instances, the computation modules on each instance converge to the same path. Also, legacy networks have embedded control and are not logically centralized, so it's eventual consistency doesn't impede it's normal operations. However, the authors of the paper admit that there could be some benefit to having some sequencing guarantees that provide deterministic state transition in the network.

(We could potentially allow for causal consistency in the graph view with our logs. Whether this is actually useful is questionable though.)

Out of the box, ONOS has support for a distributed controller architecture. It can run on multiple servers, where each is the master OpenFlow controller for some switches, and any switch has exactly one master controller. Therefore, an ONOS instance must propagate state changes from the switches in its domain to the network of ONOS controllers. The system also aims to be scalable so that as data plane capacity grows or control plane demand increases, more ONOS instances can be created.

ONOS uses Zookeeper as a coordination mechanism for general fault tolerance and master election for switches. Since ONOS is distributed, it has secondary controllers for each switch in case the primary controller goes down. That way, the load of a primary failure are distributed to the remaining controllers. During runtime failure of any nodes detected by the lack of a keep-alive message, a replacement leader or follower is elected to replace it.

Since ONOS assigns a switch to multiple instances, and exactly one master, it also uses a master election process for failure handling. The switch's master connection communicates all the relevant information about the switch to the network view, and the secondary controllers for the switch are only required to elect a new master. The authors emphasize that using Zookeeper to coordinate this election ensures that switches always have a unique master.

The data model behind the graph abstraction represents each network object in a different table. This was implemented as a means to reduce the number of updates, since a previous version stored all the network objects in a single table.

(We could store elements of the network in separate logs, since they don't have dependencies among each other)

(An issue with using a fuzzy log in this system to represent network state is that polling is inefficient. The authors found that they needed to build an event notification system so that the central database wouldn't need to be polled for changes. In a fuzzy log situation, reads require going to the central database, so it may be less efficient.)

¡onos diagram¿

The Network View API

2.2 OpenDaylight

The OpenDaylight defines a model driven approach for managing software defined networks. It aims to create a more modular system for SDN, where subscribers and providers for services are plugins to the system. This Model Driven Software Engineering approach creates a framework for relationships between models, standard mappings and patterns that enable model generation. An advantage of the system, therefore, is that it can create code and API's directly from models.

OpenDaylight controller uses YANG as a data modeling language to specify these cross platform portable data types, which can describe network operation. While YANG was initially developed for this purpose, it effectively describes other network elements, including policy, protocol, and subscriptions to services. Rather than being an object-oriented structure, YANG is tree-structured and includes complex data types like lists and unions.

For example to represent a grouping of nodes in YANG:

```
grouping target {
    leaf address {
        type inet:ip-address;
        description ”Target IP address”;
    }
    leaf port {
        type inet:port-number;
        description ”Target port number”;
    }
}
container peer {
    container destination {
        uses target;
    }
}
```

To modify the data defined by YANG, OpenDaylight uses the NETCONF and RESTCONF protocols. NETCONF is a network management protocol that defines the configuration and operations on a data store. It also has the ability to run remote procedure calls and send notifications about the underlying data. NETCONF encodes its messages and data in XML. RESTCONF is similar to NETCONF, except that it exports a REST interface over HTTP for use by external applications.

An operation configuring such a peer with NETFCONF:

```
<peer>
  <destination>
```

```
          <address >192.0.2.1</address>
          <port >830</port>
        </destination >
      </peer>
```

¡opendaylight schematic¿

The infrastructrue of

Service abstraction layer Network devices and applications go through plugins. Applications interact with the data store through RESTCONF Plugins can be binding aware or binding independednt Datastore can be pugged into any implementation Can get topology information from a bgp feed

Plugins? jpplications: Bgp and topology Model to model Generic model adoption

OpenDaylight network virtualization

[4]

2.3 Comparison of SDN systems

¡chart?¿

¡The OpenDaylight controller is built using the Java OSGi framework. OSGi adds a modular framework that allows starting, stopping, loading and unloading of Java modules without bringing down the entire running JVM platform. Ryu controller does not offer this comodity. In Ryu we need to stop the controller and run it again with the needed modules for the REST methods we want to execute or we need to build a REST API with all REST modules included in it, which runs once and provides all REST methods without stopping the controller. ¿

how is topology discovered

3 Common Data Structures

ugh

4 Proposed SDN extensions

There are two main representations of state in the systems we have discussed. There is a coordination service ONOS that maintains the naming and other information that must be stored by the controllers. Also, there is a representaion of the graph topology that each controller must learn from the physical switches under its domain. Then, to generate the full topology of the network, these subnetwork topologies must be assembled in the full network view.

4.1 Georeplication

In both ONOS and OpenDaylight, the graph topology information stored by the nodes is eventually consistent. The ONOS paper suggests that this is ”good enough” for traditional netowrking, and presumably good enough for their applications. However, I would argue that this can lead to problems with advanced SDN applications, primarily with network virtualization.

Imagine a network configuration with two groups that must be kept separate—Imagine that North and South Korea are on the same SDN network, and we want no packets to flow between the North Korea Virtual Network Segement and the South Korea VNS. In this case, imagine the following occurs:

Event 1 Switch A connects to a controller in the South Korea VNS.

Event 2 Switch A connects to a controller in the North Korea VNS.

Event 3 Switch A disconnects to the the South Korea VNS controller.

Instead, with eventual consistency over the topology, the following ordering of events could be observed:

Event 1 Switch A connects to a controller in the South Korea VNS.

Event 3 Switch A disconnects to the South Korea VNS controller.

Event 2 Switch A connects to a controller in the North Korea VNS.

Which the virtualization service deems valid after Event 3.

Therefore, before it is possble for the network virtualizaton application to observe after the disconnection event that the Switch is disconnected from the South Korean VNS. In reality, however, the switch is connected to both the South Korean and North Korean VNS's. And if the As we can see, eventual consistency leaves something to be desired when virtualzing a network.

Is this a global invariant that any switch cannot be part of the South and North Korea VNS's? If so, supporting causal consistency of the graph operations wouldn't do much to prevent such an attack on the system.

If the graph supported causal ordering guarantees, we could know that

4.2 Coordination

Different netowrk elements only work with their own data.

4.3 ONOS

make graph structure causally consistent (topology representations with a log backing...)

build a log based event notification system?

causal graph system eventual consistency does not seem to affect the control plane behavior in traditional applications

adjacency matrix of the graph in the logs

causal coordination system since there are diffrent network elements

maintian separations of the topology graph with adjancency in the runtime

4.4 OpenDaylight

OpenDaylight's modularity and data store agnostic setup makes it well suited for weakly consistent structures.

5 Conclusion

References

[1] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 325–340.

[2] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., ET AL. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), ACM, pp. 1–6.

[3] JAIN, R., AND PAUL, S. Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE 51*, 11 (November 2013), 24–31.

[4] MEDVED, J., TKACIK, A., VARGA, R., AND GRAY, K. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a* (June 2014), pp. 1–6.

[5] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[6] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 309–324.