2017 International Conference on Computational Science and Computational Intelligence

Formalizing Data Management Systems: a Case Study of Syndicate Protocol

Chun-Kun Wang
Department of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
Email: amos@cs.unc.edu

Hao Xu
Data Intensive Cyber Environment Center
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
Email: xuh@cs.unc.edu

***Abstract*—The large volume of data delivers valuable insights to many fields. A variety of big data database options have emerged for the purpose of data management. Many legacy data management systems, however, are written without formalization. Syndicate is served as a distributed file system that builds a coherent storage abstraction from already-deployed commodity components, including cloud storage and dataset providers. Through Syndicate, users can define their own provider-agnostic storage functionality to access different databases. Syndicate that fully decouples applications from underlying components generalizes the use of data management systems. It is also driving the need for further specification to guarantee the consistency and the correctness, such as functional requirements, data consistency, access control, and fault tolerance. In this paper, we take initial steps to formalize the Syndicate protocols with the goal of providing a general solution by using formal methods to improve the quality of data management systems.**

***Index Terms*—Distributed file system, Concurrent/Parallel program, π-calculus.**

I. Introduction

The large volume of data delivers valuable insights to many fields. A variety of big data database options have emerged for the purpose of data management. Many legacy data management systems, however, are written without formalization. With increasing use of the cloud, the growing size and complexity of distributed file systems are driving the need for further specification toward verifying the consistency and the correctness.

Syndicate [6] is a scalable software-defined storage system for wide-area networks. It creates virtual cloud storage volumes on top of already-deployed commodity systems, but while preserving end-to-end domain-specific storage invariants. In doing so, Syndicate fully decouples applications from providers, allowing applications to implement domain-specific storage functionality in a provider-agnostic way. Syndicate generalizes the use of data management systems so that it encounters several storage design challenges, including functional requirements, data consistency, access control and fault tolerance. The provider-agnostic programming model also makes Syndicate a platform for the generalization of data management systems. The prototype of Syndicate currently supports Amazon S3, Dropbox, iRODS, Cyverse Datastore, FTP Server and local disk.

Mechanical verification of programs is the only known way to guarantee that a software is free of programming errors. Among the tools for such verification, interactive theorem proving is not limited to specific properties or finite state spaces, compared to other automated tools such as static analysis and model checkers. Interactive proof assistants require inductive reasoning that can handle infinite state space systems directly. This paper adopts a theorem prover to specify the distributed file system.

Regarding concurrency, programs cannot be verified by a theorem prover, if they use programming constructs of which the proof assistant is unaware. Applπ library built on top of the Coq proof assistant is designed to enable concurrent programs to be modeled and verified. This library contains a formalization of a concurrent language based on π-calculus, and is based on spatial logic and a collection of lemmas for formal verification. π-calculus is an extension of the process algebra CCS and is a foundational language for the study of concurrent systems. It allows channel names to be communicated along the channels themselves. In this paper, we describe an ongoing effort to formalize the Syndicate protocols with the goal of providing a general solution by using formal methods to improve the quality of data management systems.

II. Related Work

Formal reasoning for file systems has been widely adopted for the purpose of verification. Many specification languages, such as Coq [1], Isabelle/HOL [2], Athena [3], Alloy [4], Prototype Verification System (PVS) [5], etc., are broadly used to certify systems software. The research on formal reasoning is summarized as follows. Arkoudas et al.(2004) proved the correctness of a rudimentary file system using Athena, excluding file permissions, dates, multi-layered directories, and cache management. Hesselink et al.(2012) verified a hierarchical file system and used PVS to prove its specification involving a UNIX-like permission mechanism. In terms of file system crashes, Crash Hoare logic was later introduced and applied to the specification of the file system, FSCQ [12], including crashes. FSCQ guarantees no data loss under any sequence of crashes followed by reboots. The seL4 project [11] is the first verified operating-system microkernel using the Isabelle proof assistant. Since microkernel does not provide high-level

978-1-5386-2652-8/17 $31.00 © 2017 IEEE
DOI 10.1109/CSCI.2017.178

abstractions over the hardware, the verification of seL4 does not include the correctness of a file system.

π-calculus was first introduced by Robin et al. [13]. It is a type of process calculus designed for representing parallel computation. Since its introduction, it has been widely used in many applications. Abadi et al. [15] extended π-calculus to have more primitives for encryption and decryption. π-calculus has also been extended to Business Process Modeling Language (BPML) [16]. Hirchi et al. [19] models RFID protocols using a process algebra and verifies security properties. In addition, session types has became a popular way to verify the protocols. The session types is a type π-calculus that describes communication protocols as a type in which programs can be checked to see if they conform to protocols either statically (at compile-time) or dynamically (at runtime).

The Applπ library [7] is introduced with specification and verification for processes of a π-calculus extension. It uses a Higher-Order Abstract Syntax representation in the Coq proof assistant. Affeldt, Reynald, and Kobayashi (2003) [10] first showed how to use the Applπ library for verifying an existing concurrent program. The techniques used in the verification for a SMTP server [8] have become the prototype of the Applπ library. Affeldt et al.(2005) [9] introduced partial order reduction for verification of spatial properties of pi-calculus processes. This work defined the Temporal-Spatial Logic and a notion of invisible communication for π-calculus. None of the works above verifies a distributed file system.

III. Syndicate architecture

Syndicate [6] is a virtual cloud storage service that builds a coherent storage abstraction from already-deployed commodity components, including cloud storage, edge caches, and dataset providers. It defines a cloud storage abstraction, called a Volume, which organizes application data across underlying storage with a logically centralized control-plane and a distributed data-plane. Syndicate consists of two components: a set of peer Syndicate Gateways (SG), and a scalable Metadata Service (MS). Syndicate's data-plane is made up of multiple SGs, which are user-programmable HTTP servers and clients that send and receive file data to one another, their clients, and back-end storage systems.

A. Syndicate gateways

SGs are the middleware processes that interface between Syndicate and the existing storage elements. Figure 1 presents the logical positioning of all Syndicate Gateways. The SG comes in three variants, such as User SGs, Replica SGs and Acquisition SGs. The difference between three SGs depends on how they interface with the outside world.

User SGs represent interfaces with user/application processes. They implement the Volume abstraction to application read/write requests, i.e. read requests from edge caches, and read/write requests from peer SGs. Replica SGs are interfaces with cloud storage providers. They respond to read/write requests from peer User SGs, but do not generate any requests of their own. Acquisition SGs are responsible for the interfaces

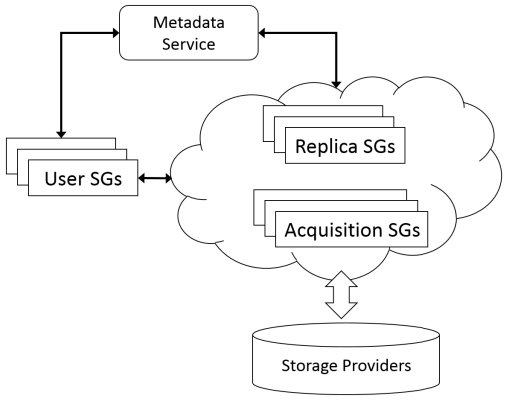

Fig. 1. Logical positioning of Syndicate Gateways (SGs).

with dataset providers. They map existing data sets into one or more Volumes as a read-only directory hierarchy. They respond to read (but not write) requests from peer SGs, but do not generate any local requests.

The SGs coordinate to meet consistency, security, and storage policies via a shared MS. This shared MS maintains the authoritative state of each Volumes metadata and helps the system scale, tolerate faults, and keep data consistent. The MS binds each SG to one Volume and helps SGs discover their peers. An application typically has one MS and places data in objects distributed into one or more Volumes. Each host runs an SG locally for each Volume to attach cloud storage and external datasets.

B. Syndicate protocol

Syndicate protocol is composed of the communication between SGs and MS. Since MS always keeps object manifests fresh and consistent, all requests have to visit MS either to get the latest manifests or update the manifests, regardless of the requests, i.e. read, write, rename, etc. For instance, the read operation needs to visit MS first in order to get the manifest that keeps track of the data object information. After finishing operations, the read operation has no need to visit MS, while the write operation has to update the manifest in the MS. The overall Syndicate protocol can be simply broken down into three parts: the gateway-to-gateway protocol, the gateway-to-MS protocol and the gateway-to-driver protocol. In this paper, we focus on the gateway-to-gateway protocol, which deals with the requests for accessing the file system from storage, without considering consistency of the manifest.

IV. Methodology

A. π-calculus

The π-calculus models the processes, parallel execution and communication between processes. We follow the notations defined at [14]. The π-calculus contains two kinds of entities: processes and channels. Processes, sometimes called agents, are the active components of a system; they interact through

synchronous channels, also called names. Two processes synchronously exchange a single data value, which could also be a channel itself. Namely, a name received on a channel can then be used itself as a channel name for output or input.

We follow the naming conventions and the syntax in which u, v, w, x, y, z range over channels and $A, B, C, \cdots$ range over agent identifiers. The abstract syntax of an agent in π-calculus is defined as follows:

$$P ::= \quad \mathbf{0} \quad | \quad x(y).P \quad | \quad \bar{x}y.P \quad | \quad (\nu x)P \quad | \quad P|Q \quad | \quad !P,$$

- $\mathbf{0}$ is termination, meaning that Agent P does not do anything.
- $(\nu x).P$ represents restriction that creates a new channel x and runs P.
- $x(y).P$ means that the free name y is bound in P along the channel named x and then behaves like P.
  Name y is said to be free, if y is not bound. For example, if y is bound to P, that means y can be only used inside of P, which y is private to P.
- $\bar{x}y.P$ means that Agent P sends the bound name y out along channel x and then behaves like P.
- $P|Q$ means that Process P and Agent Q execute in parallel. P and Q may behave independently or they may interact with each other.
- $!P$ is called replication and can be thought of as an infinite composition $P|P|P|\cdots$.

The labeled semantics of the π-calculus is defined by a set of transition rules which establish the system evolution. A transition rule is in the form of $P \xrightarrow{\alpha} P'$, which means that process P evolves to P' after performing a computation step $\xrightarrow{\alpha}$. In labeled semantics of the π-calculus, α is the prefix that represents four kinds of actions as follows:

$$\begin{array}{rrl} \alpha ::= & \tau & \text{silent action} \\ | & x(y) & \text{input action} \\ | & \bar{x}y & \text{output action} \\ | & \nu x & \text{create action} \end{array}$$

The prefix $x(y)$ means input some name along the link named x and call it y. It is also called *bound input* in which brackets act as formal binder. For instance, prefix $x(y)$ in $x(y).P$ binds the free occurrences of y in P. The prefix $\bar{x}y$ means output the free name y along the link named x. It is also called *free output*. The create action has a prefix, νx, that represents the creation of a new link named x. Besides the prefix $x(y)$, νx is another formal binder, the restriction operator νy in $\nu y.P$. A restricted name can be sent outside its original scope.

The main reduction rule captures the ability of processes to communicate through channels. For example, $\bar{x}z.P|x(y).Q \xrightarrow{\bar{x}z} P|Q[z/y]$, where $Q[z/y]$ denotes the process Q in which the free name z has been substituted for the free occurrences of y, means that Process P sends name z out along the channel x, and Process Q receives the value z along the channel x and substitutes for the name y. More explanations of reduction rules can be found in [14], [17].

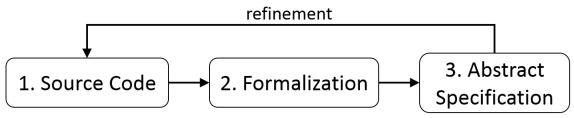

Fig. 2. Three stages of the specification for Syndicate protocol.

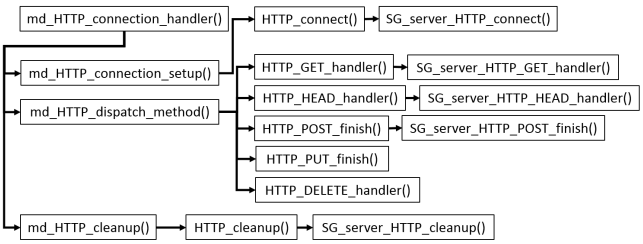

Fig. 3. The code diagram of the subsystem of Syndicate connection handler.

B. Formalization

This section introduces the formalization for the Syndicate protocol. Figure 2 presents the specification overview of Syndicate protocol. First of all, the Syndicate source code, written in C++, Python and Java, is traced by code tracing tools. Code tracing tools generate the code diagrams, which are modeled by π-calculus in the second stage. Figure 3 shows a code diagram example of the connection handler. The formalization stage contains all data structure and implementation details for the next stage. The abstract specification is the main specification for Syndicate protocol. The concurrent primitives that the Applπ library provided for π-calculus logic enable concurrent programs to check the correctness. A basic file system is also built and specified in Coq. Finally, we are able to refine the source code or the coverage of code diagrams.

The whole communication of the Syndicate gateway-to-gateway protocol can be abstracted as three components, such as client, server and file system. Abstraction of client, server and file system supports the more generic expression of the specification. The client is responsible for sending a request to the server and waiting for the response from the server. The client is formalized by π-calculus as follows:

$$Client: \quad \bar{C}\, req\, .\, C\, (resp)\, .\, 0$$

Upon receiving the request from clients, the server requires a random number from the channel $Rand$ and selects a corresponding file system worker that provides services to access the distributed file system. It waits for the response from the file system worker and returns the response back to the client. The π-calculus formalization of the server is as follows:

$$\begin{aligned} Server: \quad & !C\, (req)\, .\, Rand\, (n)\, .\, \bar{C_{fn}}\, freq(req)\, . \\ & C_{fn}\, (fresp)\, .\, \bar{C}\, resp(fresp)\, .\, 0 \end{aligned}$$

The file system worker receives the request from the server. It requires the current file system state from the channel F_{st} and accesses the file system. After modifying the file system, a

new file system state may be generated. The worker preserves the updated file system state by sending it to the channel F_{st} and generates the response for the server. The way to preserve the file system state by using a channel is inspired by the work [18]. A file system worker is formalized as follows:

$$FS: \quad !C_{fn}(freq) \,.\, F_{st}\,(Fst) \,.\, \bar{F_{st}}\, Fst(freq, Fst) \,. \\ \bar{C_{fn}}\; fresp(freq, Fst) \,.\, 0$$

The Syndicate client, server and file system worker are formalized by π-calculus and then encoded by Applπ library. We translate the π-calculus formalization by using Applπ library. The complete formalizations and lemmas are shown in the GitHub; see the link, https://github.com/ChunKunWang/syndicate-core-logic.

V. Conclusion

Syndicate is served as a distributed file system that builds a coherent storage abstraction from already-deployed commodity components. It fully decouples applications from providers, allowing applications to implement domain-specific storage functionality in a provider-agnostic way. With the provider-agnostic programming model, Syndicate is able to generalize the use of data management systems. In this paper, we describe an ongoing effort to formalize the Syndicate protocols with the goal of providing a general solution to improve the quality of data management systems. We adopts π-calculus to formalize Syndicate protocol and shows how the Applπ library specifies the Syndicate client, server and file system.

This paper is a preliminary work regarding the specification of one part of the Syndicate protocol. We haven't finished the specification for all Syndicate components, including CDNs, Metadata Service, Edge cache, etc., and haven't touched the gateway-to-MS and the gateway-to-driver protocols. More consideration must be given to the complete specification for Syndicate as future work.

Acknowledgment

This paper is partially based upon work supported by the National Science Foundation under Grant Number OAC 1541318 "CC*DNI DIBBs: Give Your Data the Edge: A Scalable Data Delivery Platform." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Chlipala, Adam. *Certified programming with dependent types.* 2011.
[2] *Isabelle/HOL*, 2016. Available at: https://isabelle.in.tum.de/dist/library/HOL/index.html
[3] *Athena*, 2016. Available at: http://www.proofcentral.org/athena/
[4] *Alloy*, 2016. Available at: http://alloy.mit.edu/alloy/faq.html
[5] S. Owre, N. Shankar, J. M. Rushby, and D., *Version 2.4 System Guide, Prover Guide*, 2001. http://pvs.csl.sri.com/
[6] Nelson, Jude C., and Larry L. Peterson. *Syndicate: virtual cloud storage through provider composition.* Proceedings of the 2014 ACM international workshop on Software-defined ecosystems. ACM, 2014.
[7] Affeldt, Reynald, and Naoki Kobayashi. *A Coq library for verification of concurrent programs.* Electronic Notes in Theoretical Computer Science 199 (2008): 17-32.
[8] Affeldt, Reynald, Naoki Kobayashi, and Akinori Yonezawa. *Verification of concurrent programs using the coq proof assistant: A case study.* IPSJ Digital Courier 1 (2005): 117-127.
[9] Affeldt, Reynald, and Naoki Kobayashi. *Partial order reduction for verification of spatial properties of pi-calculus processes.* Electronic Notes in Theoretical Computer Science 128.2 (2005): 151-168.
[10] Affeldt, Reynald, and Naoki Kobayashi. *Formalization and verification of a mail server in Coq.* Software SecurityTheories and Systems. Springer Berlin Heidelberg, 2003. 217-233.
[11] Klein, Gerwin, et al. *seL4: Formal verification of an OS kernel.* Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.
[12] Chen, Haogang, et al. *Using Crash Hoare logic for certifying the FSCQ file system.* Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015.
[13] Milner, Robin, Joachim Parrow, and David Walker. *A calculus of mobile processes, i.* Information and computation 100.1 (1992): 1-40.
[14] Milner, Robin. *The polyadic π-calculus: a tutorial.* Logic and algebra of specification. Springer Berlin Heidelberg, 1993. 203-246.
[15] Abadi, Martn, and Andrew D. Gordon. *A calculus for cryptographic protocols: The spi calculus.* Proceedings of the 4th ACM conference on Computer and communications security. ACM, 1997.
[16] Smith, Howard. *Business process managementthe third wave: business process modelling language (bpml) and its pi-calculus foundations.* Information and Software Technology 45.15 (2003): 1065-1069.
[17] Quaglia, Paola, and BRICS Lecture Series LS. *The-calculus: Notes on labelled semantics.* (1998).
[18] Aranda, Jess, et al. *On recursion, replication and scope mechanisms in process calculi.* Formal Methods for Components and Objects. Springer Berlin Heidelberg, 2007.
[19] Hirschi, Lucca, David Baelde, and Stphanie Delaune. *A method for verifying privacy-type properties: the unbounded case.* Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 2016.