RChain Architecture Documentation

Release 0.9.0

Ed Eykholt, Lucius Gregory Meredith, Joseph Denman

Oct 28, 2018

Contents:

Authors Ed Eykholt, Lucius Meredith, Joseph Denman

Date 2017-07-22

Organization RChain Cooperative

CHAPTER 1

Notice: Updated Details Available

For details on the platform as it is built, see developer.rchain.coop including rchain/rchain open source github repository, Rholang tutorial, and project status.

CHAPTER 2

Abstract

The RChain Platform Architecture description provides a high-level blueprint of the RChain decentralized, economically sustainable public compute infrastructure. While the RChain design is inspired by that of earlier blockchains, it also realizes decades of research across the fields of concurrent and distributed computation, mathematics, and programming language design. The platform includes a modular, end-to-end design that commits to correct-by-construction software and industrial extensibility.

Intended audience: This document is written for software developers and innovators who are interested in decentralized systems.

2.1 Motivation

The decentralization movement is ambitious and will provide awesome opportunities for new social and economic interactions. Decentralization also provides a counterbalance to abuses and corruption that occasionally occur in large organizations where power is concentrated. Decentralization supports self-determination and the rights of individuals to self-organize. Of course, the realities of a more decentralized world will also have their challenges and issues, such as how the needs of international law, public good, and compassion will be honored.

We admire the awesome innovations of Bitcoin, Ethereum, and other platforms that have dramatically advanced the state of decentralized systems and ushered in this new age of cryptocurrency and smart contracts. However, we also see symptoms that those projects did not use the best engineering and formal models for scaling and correctness in order to support mission-critical solutions. The ongoing debates about scaling and reliability are symptomatic of foundational architectural issues. For example, is it a scalable design to insist on an explicit serialized processing order for all of a blockchain's transactions conducted on planet earth?

To become a blockchain solution with industrial-scale utility, RChain must provide content delivery at the scale of Facebook and support transactions at the speed of Visa. After due diligence on the current state of many blockchain projects, after deep collaboration with other blockchain developers, and after understanding their respective roadmaps, we concluded that the current and near-term Blockchain architectures cannot meet these requirements. In mid-2016, we resolved to build a better blockchain architecture.

Together with the blockchain industry, we are still at the dawn of this decentralized movement. Now is the time to lay down a solid architectural foundation. The journey ahead for those who share this ambitious vision is as challenging as it is worthwhile, and this document summarizes that vision and how we seek to accomplish it.

2.2 Approach

We began by admitting the following minimal requirements:

- Dynamic, responsive, and provably correct smart contracts
- Concurrent execution of independent smart contracts
- Data separation to reduce unnecessary data replication of otherwise independent tokens and smart contracts
- Dynamic and responsive node-to-node communication
- Computationally non-intensive consensus/validation protocol

Building quality software is challenging. It is easier to build "clever" software; however, the resulting software is often of poor quality, riddled with bugs, difficult to maintain, and difficult to evolve. Inheriting and working on such software can be hellish for development teams, not to mention their customers. When building an open-source system to support a mission-critical economy, we reject a minimal-success mindset in favor of end-to-end correctness.

To accomplish the requirements above, our design approach is committed to:

- A computational model that assumes fine-grained concurrency and dynamic network topology;
- A composable and dynamic resource addressing scheme;
- The functional programming paradigm, as it more naturally accommodates distributed and parallel processing;
- Formally verified, correct-by-construction protocols which leverage model checking and theorem proving;
- The principles of intension and compositionality.

2.3 Introduction

The open-source RChain project is building a *decentralized, economic, censorship-resistant, public compute infrastructure and blockchain*. It will host and execute programs popularly referred to as "smart contracts". It will be trustworthy, scalable, concurrent, with proof-of-stake consensus and content delivery.

Using smart contracts, a broad array of fully-scalable decentralized applications (dApps) can be built on the top of this platform. DApps may address areas such as identity, tokens, timestamping, financial services, monetized content delivery, Decentralized Autonomous Organizations (DAOs), exchanges, reputation, private social networks, marketplaces, and many more.

The RChain Network implements direct node-to-node communication, where each node runs the RChain platform and a set of dApps on the top of it.

The heart of an RChain is the Rho Virtual Machine (RhoVM) Execution Environment, which runs multiple RhoVMs that are each executing a smart contract. These execute concurrently and are multi-threaded.

This concurrency, which is designed around on the formal models of mobile process calculi, along with an application of compositional namespaces, allows for what are in effect *multiple blockchains per node*. This multi-chain, independently executing virtual machine instances is in sharp contrast to a "global compute" design which constrains transactions to be executed sequentially, on a single virtual machine. In addition, each node can be configured to subscribe to and process the namespaces (blockchains) in which it is interested.

Like other blockchains, achieving consensus across nodes on the state of the blockchain is essential. RChain's protocol for replication and consensus is called Casper and is a proof-of-stake protocol. Similar to Ethereum, a contract starts out in one state, many nodes receive a signed transaction, and then their RhoVM instances execute that contract to its next state. An array of node operators, or "bonded validators" apply the consensus algorithm to crypto-economically verify that the entire history of state configurations and state transitions of the RhoVM instance are accurately replicated in a distributed data store.

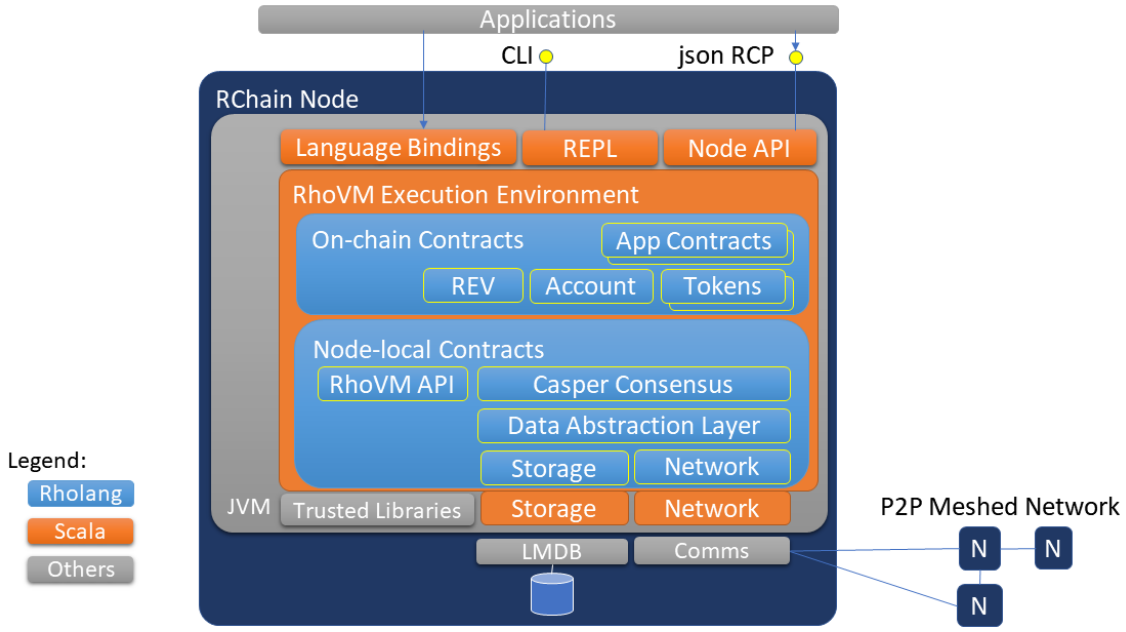


Fig. 1: Figure: High-level RChain Architecture

The blockchain contracts (aka smart contracts, processes, or programs), including system contracts included in the installation are written in the RChain general-purpose language "**Rholang**" (Reflective higher-order language). Derived from the rho-calculus computational formalism, Rholang supports internal programmatic concurrency. It formally expresses the communication and coordination of many processes executing in parallel composition. Rholang naturally accommodates industry trends in code mobility, reactive/monadic API's, parallelism, asynchronicity, and behavioral types.

Since nodes are internally concurrent, and each need not run all namespaces (blockchains), the system will be *scalable*.

Since the contract language and its VM are build from the formal specifications of provable mathematics, and since the compiler pipeline and engineering approach is *correct by construction*, we expect the platform will be regarded as *trustworthy*.

2.4 Comparison of Blockchains

The following table compares several key qualities of Bitcoin, Ethereum, and RChain.

2.5 Architecture Overview

The primary components of the architecture are depicted below:

The execution architecture may rely on some operating-specific external components, but these are kept to a minium by running on the JVM. The RhoVM Execution Envirionment runs on the JVM, and then the individual RhoVM instances run within the RhoVM Execution Environment.

The **RhoVM Execution Environment** provides the context for contract execution, the lifecycle of individual RhoVM instances.

Describing the remaining layers depicted, from the bottom-up:

This document assumes the reader is familiar with the basics of Bitcoin and Ethereum. As one approach to introducing the architecture let's compare the characteristics of Bitcoin, Ethereum, and RChain as currently planned.

		Bitcoin	Ethereum	RChain
Semantic Data Structure		Blockchain -- a chain of blocks. Each block contains a header that points at the previous block, a list of transactions, and other data.	Blockchain -- a chain of blocks. Each block contains a header that points at the previous block, a transaction list, and a ommers (uncles) list.	Blockchain -- a graph of blocks. Each block contains a header that points at one or more previous blocks, a list of transactions, and other metadata.
Consensus	*Algorithm*	Proof of work	Current: Proof of work. Future: Proof of Stake -- stake-based betting on blocks.	Proof of Stake: Stake-based betting on logical propositions.
	Finality	Probability of transaction reversal diminishes over time, at each new block confirmation.	Probability of transaction reversal diminishes over time, at each new block confirmation.	Probability of transaction reversal diminishes over time, at each new block confirmation.
	Visibility	Global	Private, consortium, or public depending on deployed nodes.	Private, consortium, or public depending on namespace and/or deployed nodes.
	Revision Mechanism	Soft and hard forks	Current: Soft and hard forks. Future: Block revisions in case of temporary network isolation.	Block revisions in case of temporary network isolation.
Address		Identifier that represents a destination for a bitcoin transaction. Derived from a random private key or from hash of a script (in P2SH)	An identifier that represents a possible destination for an Ethereum transaction, derived from a random private key to represent an account.	An encapsulated, encrypted, and unique channel to communicate with a process (including a smart contract), similar to a URL.
Sharding	*Heterogeneity*	Homogeneous, i.e., not sharded	Current: Homogeneous, i.e., not sharded Future: two-level	Isolated address "namespaces" allow clients to subscribe to selected address without downloading the entire blockchain. May impose different security policies on different address spaces.
	Basis for sharding	N/A	Address range	Dynamic composable sharding based on namespace interaction.
	Number of levels	N/A	Future: two levels: cluster + leaves	Levels of arbitrary depth and size.
	Concurrency	N/A	Current: No Future: Asynchronous. Not parallel.	Yes. Language allows asynchronous and parallel execution of contracts. VM allows parallel execution of many VMs -- Concurrent validation

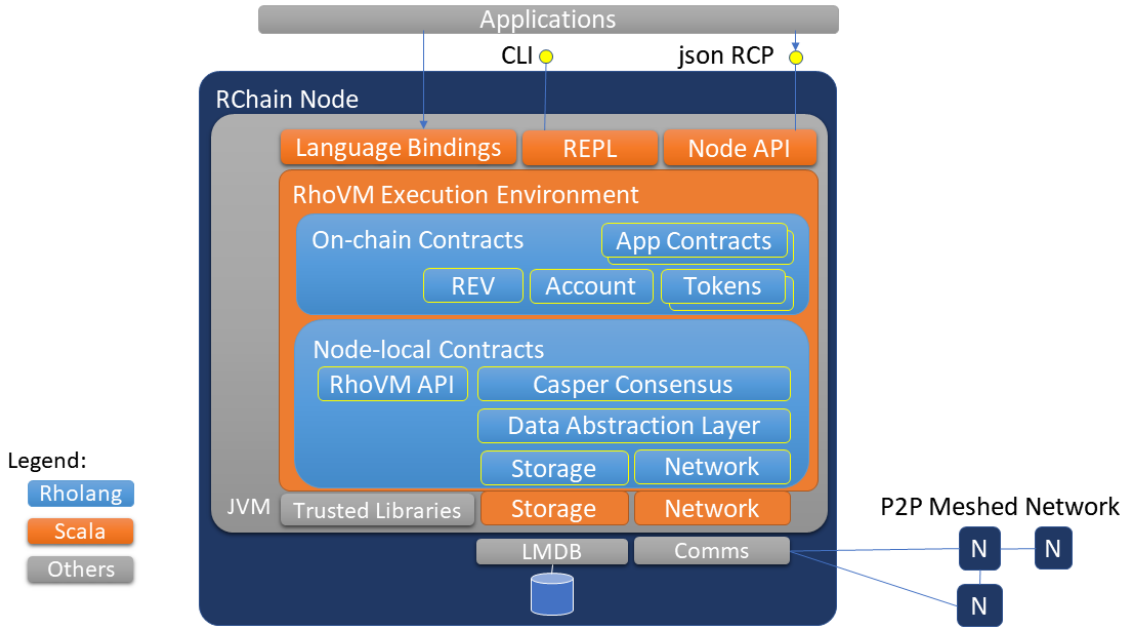


Fig. 2: Figure - The RChain Architecture

P2P Communication supports node-to-node communications. This will be a TBD commercial-grade, open-source component such as ZeroMQ or RabbitMQ.

Storage is via MongoDB, a key-value datastore. The primary in-memory data structure is a radix tree (trie).

Data Abstraction Layer provides monadic access to data and other nodes consistently, as if they were local. This layer is an evolution of the SpecialK technology (including its decentralized content delivery, key-value database, inter-node messaging, and data access patterns). This layer is being implemented in Rholang, and so it relies on the RhoVM-EE and Rholang's **Foreign Function Interface** to access P2P Communication and Storage.

Consensus (Casper Proof-of-Stake validation/consensus Protocol) assures node consensus on the state of each blockchain.

All RChain nodes include essential **System Contracts**, which are written in Rholang. System processes include those for running RhoVM instances, load balancing, managing dApp contracts, tokens, node trust, and others.

The Token system contracts include those required to run protocols that interact beyond the local node. These are *protocol acce*

- **Staking tokens** are those required to run consensus, including the **RChain Rev** token. Additional staking tokens may be introduced through official software releases. A staking token is required to pay for node *resources*. **Phlogiston** is RChain's measure of the cost of resources (similar to *gas* in Ethereum), and it is multi-dimensional and depends on usage of compute (depending on instruction), storage (depending on size and duration), and bandwidth (quality-of-service and throughput) resources. See also section entitled "Rate-limiting Mechanism."
- **Application tokens** are optional and may be required to run certain dApps. New application tokens can be introduced at any time by a dApp developer, and are similar to Ethereum's ERC20 tokens.

The **Rho API** provides access to Execution Environment and the Node. **Language Bindings** will be available for programming languages written against the JVM, and potentially others. A **REPL** (Read, Execute, Print, and Loop) development tool will be provided. Each node will have a **Command Line Interface** CLI. A **Node API** will expose features via HTTP and json RPC.

2.5.1 Concurrency vs. Parallelism

It is essential the reader understands the implications of concurrent execution. When we say, "concurrency", we are not referring to the simultaneous execution of multiple processes. That is parallelism. *Concurrency* is a structural property which allows independent processes to compose into complex processes. Processes are considered independent if they do not compete for resources.

Since RChain has committed to concurrency in Rholang and RhoVM, we'll see that we will get parallelism and asynchronicity as "free" emergent properties. Whether the platform is running on one processor or 1,000,000 processors, the RChain design is scalable. Having said that, the reader of this document will notice design patterns of concurrent computation throughout.

2.6 Node and Blockchain Semantics

The following UML class diagram depicts the primary conceptual classes and structural relationships.

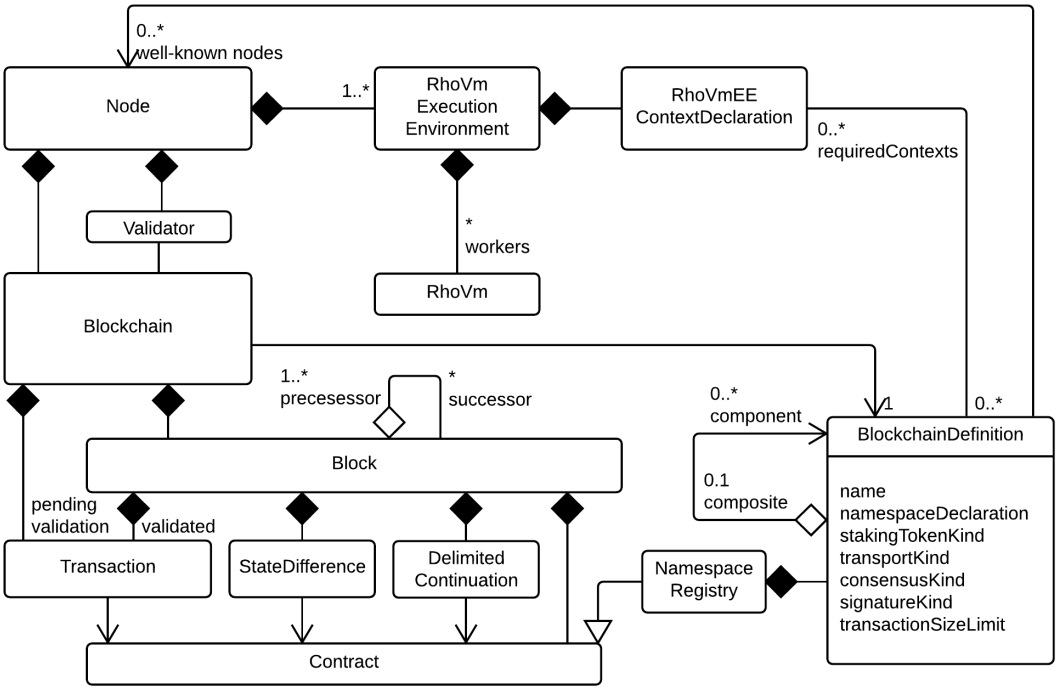


Fig. 3: Figure - RChain Blockchain Structural Semantics

2.7 Contract Design

An RChain contract is a well-specified, well-behaved, and formally verified program that interacts with other such programs.

In this section, we cover contract interaction through the production of Rholang. To begin, we give an overview of contract interaction on the RChain platform. Afterwards, we describe the core formalism RChain uses to achieve formal verification and to model concurrency on many of RChain's system layers. Then, we explore how that core model

extends to accomodate best-in-industry surface-language standards such as reflection, parallelism, asynchronicity, reactive data streams, and compile-time security-type checks.

2.7.1 Contract Overview

Used loosely as 'contract', **a smart contract is a process with:**

1. Persistent state
2. Associated code
3. Associated RChain address(es)

Important to remember is that a smart contract is of arbitrary complexity. It may refer to an atomic operation or to a superset of protocols which compose to form a complex protocol.

A contract is triggered by a message from an external network agent, where an external agent may be a contract or a network user.

A Message:

1. Is issued over a named channel(s), which may be public or private.
2. May be typed and may range in format from a simple value to an unordered array of bytes, to a variable, to a data structure, to *the code of a process*, and most things in between.

An Agent sends and receives messages on named communication links known as 'named channels'.

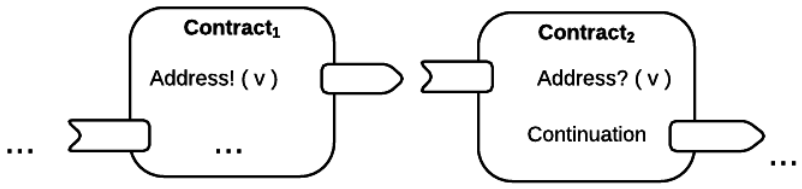
A Named Channel:

1. Is a "location" where otherwise independent processes synchronize.
2. Is used by processes to send and receive messages between each other.
3. Is provably unguessable and anonymous unless deliberately introduced by a process.

A channel is implemented as a variable that is shared between a "read-only" and a "write-only" process. Therefore, the functionality of a channel is only limited by the interpretation of what a variable may be. As a channel represents the abstract notion of "location", it may take different forms. For our early interpretation, a named channel's function may range from the local memory address (variable) of a single machine, to the network address of a machine in a distributed system.

Consistent with that interpretation, **a blockchain address is a named channel**, i.e., a location(s) where an agent may be reached.

Two contracts sending and receiving a message on the channel named 'Address':

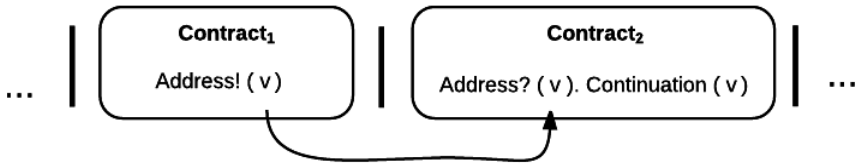


This model depicts two contracts, both of which may receive and send messages. At some point, an external actor prompts `Contract1` to send a value, `v`, on the channel, `address`, which is the address of `Contract2`. Meanwhile, `Contract2` listens on the `address` channel for some value `v`. After it receives `v`, `Contract2` invokes a process continuation with `v` as an argument. These last two steps occur sequentially.

Note that, this model assumes that at least the sender possesses the address of `Contract2`. Also note that, after it sends `v`, `Contract1` has been run to termination, thus it is incapable of sending anything else unless prompted.

Similarly, after it invokes its continuation, `Contract2` has been run to termination, thus it is incapable of listening for any other messages.

RChain contracts enjoy fine-grain, internal concurrency, which means that these processes, and any processes that are not co-dependent, may be placed in parallel composition. So, we amend our notation:



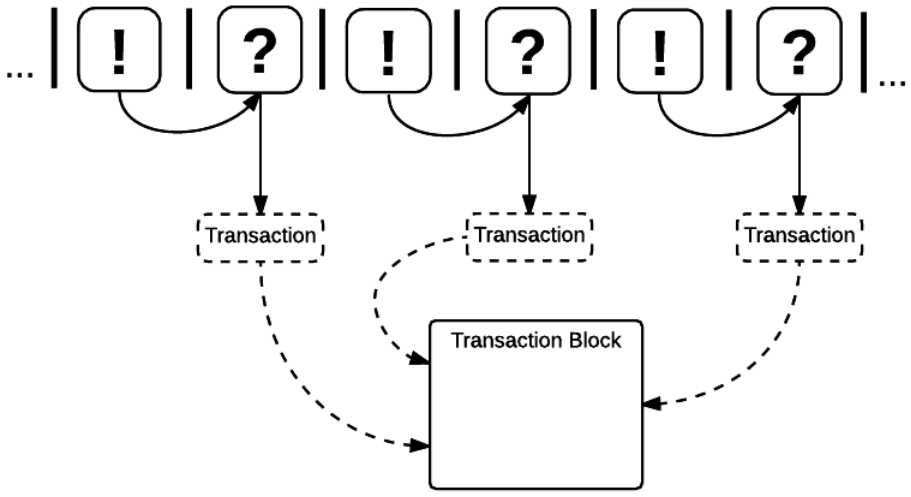
Executing in parallel with a number of other processes, an external actor prompts `Contract1` to send a value, `v`, on the channel `address` i.e. the address of `Contract2`. If `Contract1` has no value to send, it blocks. If `Contract2` has not received a value, it blocks and the continuation is not triggered.

Transactions

How do transaction semantics fit into our description of contracts? **From the process level, a transaction is an acknowledgment that a message has been "witnessed" at a channel**

Messages themselves are virtual objects, but the pre-state and post-state of a contract, referring to the states before and after a message is sent by one agent and witnessed by another, are recorded and timestamped in storage, also known (in a moral sense) as the "blockchain".

Message passing is an atomic operation. Either a message is witnessed, or it is not, and only the successful witnessing of a message qualifies as a verifiable transaction that can be included in a block. Examples hitherto depict atomic protocols, but full-bodied applications may spawn, send, and receive on tens of thousands of channels at runtime. Hence, when the value of some resource is altered and witnessed by a process, there is record of when and where it was witnessed by what agent. This implementation is consistent with an interpretation of data as a linear resource.



The ability to place a message at either end of a channel before and after the message is sent, and therefore to view the serialized form of messages, is an attribute specific to RChain. Additionally, by stating successful messages as transactions, all messages, whether from external user to contract or between contracts, are accounted for. Thus, we balance the extensible autonomy of contracts with accountability.

For an example of how this model is adaptable to industry trends in reactive programming, observe the following two contracts, which model interaction over "live" data feeds:

Executing in parallel composition with a number of other processes, `Contract1` is prompted to send a set of values, `vN`, on the channel `address` i.e. the address of `Contract2`. In this scenario, the reader will notice `Contract2` as a thread which listens for a set of values as input from a single data stream that is dual to a set of values being output from a stream at its tail. When the set of values, `v1...vN`, is witnessed at the channel, `address`, a continuation is invoked with `v1...vN` as an argument. While the interaction between `Contract1` and `Contract2` is asynchronous, the input operation `address?(v1...vN)` and `Continuation(v)` of `Contract2` are necessarily sequential. `address?(v1...vN)` is said to "pre-fix" `Continuation(v)` in every instance.

We have presented a very basic depiction of concurrent contract interaction on the RChain platform to include contracts, recognizing addresses as channels of communication, and transactions as the successful transmission of a message over said channels. Next, we outline the core system which formally models these constructs.

2.7.2 The Formalism: Rho-Calculus

Formal verification is the *de facto* standard for many mission-critical technologies. Some of the earliest formal verification methods were applied to the two-level shutdown systems of nuclear generators[1]. Many ATM software solutions verify performance by deriving solutions from models of linear temporal logic. Many military information and decision systems invoke Hoare logic to verify crash tolerance. An indiscriminate smart-contracting utility that desires to host mission-critical contracts bears the same responsibility of verifiability to its users. Therefore, our design approach to the surface-language and execution model is based on a provably correct model of computation[2].

At the same time, there are relatively few programming paradigms and languages that handle concurrent processes in their core model. Instead, they bolt some kind of threading-based concurrency model on the side to address being able to scale by doing more than one thing at a time. By contrast, the Mobile process calculi provide a fundamentally different notion of what computing is. In these models, computing arises primarily from the interaction of processes. The ability to formally verify an execution model, and to allow that execution model to be fundamentally concurrent, is why we have chosen a process calculus for RChain's model of computation.

Specifically, **the RChain execution model is derived from the syntax and semantics of rho-calculus.** The rho-calculus is a variant of the π-calculus that was introduced in 2004 to provide the first model of concurrent computation with reflection. "Rho" stands for reflective, higher-order.

Though an understanding of the π-calculus isn't necessary for the purposes of this document, those unfamiliar with the π-calculus are strongly encouraged to explore it. The π-calculus is the first formal system to successfully model networks where nodes may regularly join and drop from the network. It assumes fine-grained concurrency and process communication i.e. two processes may be introduced by a third process. The rho-calculus extension inherits all of those features and adds reflection.

For more information, see The Polyadic Pi-Calculus and Higher Category Models of the Pi-Calculus.

[1] Lawford, M., Wassyng, A.: Formal Verification of Nuclear Systems: Past, Present, and Future. Information & Security: An International Journal. 28, 223–235 (2012).

[2] In addition to selecting a formally verifiable model of computation, are investigating a few verification frameworks such as the K-Framework to achieve this.

Reflection

Reflection is now widely recognized as a key feature of practical programming languages, known broadly as "meta-programming". Reflection is a disciplined way to turn programs into data that programs can operate on and then turn the modified data back into new programs. Java, C#, and Scala eventually adopted reflection as a core feature, and even OCaml and Haskell have ultimately developed reflective versions[3]. The reason is simple: at industrial scale, programmers use programs to write programs. Without that computational leverage, it would take too long to write advanced industrial scale programs.

Syntax and Semantics

The rho-calculus constructs "names" and "processes". Similar to the π-calculus, **a name may be a channel of communication or a value. However, with the rho-calculus addition of 'reflection', a name may also be a 'quoted' process, where a quoted process is the code of a process.** The genericity of names will become important in the coming sections.

From the notion of names and processes, the calculus builds a few basic "processes". A process may have persistent state but does not assume it. The term "process" is the more general term for "smart contract". Hence, every contract is a process but not every process is smart contract.

Rho-calculus builds the following basic terms to describe interaction among processes:

```
P,Q,R ::= 0                  // nil or stopped process

        |   for( ptrn1 <- x1; ... ; ptrnN <- xN ).P // input guarded process
        |   x!( @Q )         // output
        |   \*x\             // dereferenced or unquoted name
        |   P|Q              // parallel composition

x,ptrn ::= @P                // name or quoted process
```

The first three terms denote I/O, describing the actions of message passing:

- `0` is the form of the inert or stopped process that is the ground of the model.
- The input term, `for( ptrn1 <- x1; ... ; ptrnN <- xN )P`, is the form of an input-guarded process, `P`, listening for a set of patterns, `ptrnN`, on a set of channels, `xN`. On receiving such a pattern, continuation P is invoked[4]. Scala programmers will notice the 'for-comprehension' as syntactic sugar for treating channel access monadically[5]. The result is that all input-channels are subject to pattern matching, which constructs an input-guard of sorts.
- The output term, `x!( @Q )`, sends the name, `@Q`, on channel, `x`. Although the name being sent on `x` may be a values, a channel, or a quoted process (which may itself contain many channels and values), our notation uses, `@Q` to reiterate the expressiveness of names.

The next term is structural, describing concurrency:

- `P|Q` is the form of a process that is the parallel composition of two processes P and Q where both processes are executing and communicating asynchronously.

Two additional terms are introduced to provide reflection:

- `@P` , the "Reflect" term introduces the notion of a "quoted process", which is the code of a process that is serialized and sent over a channel.
- `x` , the "Reify" term, allows a quoted process to be deserialized from a channel.

[3] See Scala Documentation: Reflection
[4] See Scala Documentation: For-Comprehensions
[5] See Scala Documentation: Delimited Continuations

This syntax gives the basic term language that will comprise the Rholang type system primitives. The rho-calculus assumes internal structure on names, which is preserved as they're passed between processes. One result of being able to investigate the internal structure of a name is that processes may be serialized to a channel and then deserialized upon being received, which means that processes may not only communicate signals to one another, they may communicate full-form processes to one another. Hence, the higher-order extension.

Rho-calculus also gives a single, reduction (substitution) rule to realize computation, known as the "COMM" rule. Reductions are atomic; they either happen, or they don't. It is the only rule which directly reduces a rho-calculus term:

```
for( ptrn <- x ).P | x!(@Q) -> P{ @Q/ptrn } //Reduction Rule
```

The COMM rule requires that two processes are placed in concurrent execution. It also requires that the two are in a co-channel relationship. That is, one process is reading from channel, `x`, while the other process is writing to the channel, `x`. The two processes are said to "synchronize" at `x`. The output process sends the quoted process, `@Q`, on `x`. In parallel, the input process waits for an arbitrary pattern, `ptrn` to arrive on `x`. Upon matching the pattern, it executes continuation `P`. After reduction, the simplified term denotes `P`, which will execute in an environment where `@Q` is bound to `ptrn`. That is, `@Q` is substituted for every occurrence of the `ptrn`, in the body of `P`.

The COMM rule implies the successful communication of a message over a channel. The reader may remember that successful communication of a message over a channel constitutes a verifiable transaction. In fact, **a reduction is a transaction** precisely because it verifies that a resource has been accessed and altered. As a result, **the number of reductions performed corresponds to the units of atomic computation performed, which are fundamentally tethered to the number of transactions committed to a block.** This correspondence ensures that all platform computation is indiscriminately quantifiable.

Another implication of being able to investigate the internal structure of a name is that channels may encapsulate yet more channels. Though they are very light in an atomic sense, when channels possess internal structure, they may function as data stores, data structures, and provably unbounded queues of arbitrary depth. In fact, in almost all implementations, a contract's persistent storage will consist of state value stored in a `state` channel which takes requests to `set` and `get` a `newValue`. We will demonstrate the wide-sweeping implications of internal structure on channels in the section on namespaces. For further details, see A Reflective Higher-Order Calculus and Namespace Logic - A Logic for a Reflective Higher-Order Calculus.

Behavioral Types

A behavioral type is a property of an object that binds it to a discrete range of action patterns. Behavioral types constrain not only the structure of input and output, but **the permitted order of inputs and outputs among communicating and (possibly) concurrent processes under varying conditions.**

Behavioral types are specific to the mobile process calculi particularly because of the non-determinism the mobile calculi introduce and accommodate. More specifically, a concurrent model may introduce multiple scenarios under which data may be accessed, yet possess no knowledge as to the sequence in which those scenarios occur. Data may be shareable at a certain stage of a protocol but not in a subsequent stage. In that sense, resource competition is problematic; if a system does not respect precise sharing constraints on objects, mutations may result. Therefore we require that network resources are used according to a strict discipline which describes and specifies sets of processes that demonstrate a similar, "safe" behavior.

The Rholang behavioral type system will iteratively decorate terms with modal logical operators, which are propositions about the behavior of those terms. Ultimately properties data information flow, resource access, will be concretized in a type system that can be checked at compile-time.

The behavioral type systems Rholang will support make it possible to evaluate collections of contracts against how their code is shaped and how it behaves. As such, Rholang contracts elevate semantics to a type-level vantage point, where we are able to scope how entire protocols can safely interface.

In their seminal paper, Logic as a Distributive Law, Mike Stay & Gregory Meredith, develop an algorithm to iteratively generate a spatial-behavioral logic from any monadic data structure.

2.7.3 Significance

This model has been peer reviewed multiple times over the last ten years. Prototypes demonstrating its soundness have been available for nearly a decade. The minimal rho-calculus syntax expresses six primitives - far fewer than found in Solidity, Ethereum's smart contracting language, yet the model is far more expressive than Solidity. In particular, Solidity-based smart contracts do not enjoy internal concurrency, while Rholang-based contracts assume it.

To summarize, the rho-calculus formalism is the first computational model to:

1. Realize maximal code mobility via 'reflection', which permits full-form, quoted processes to be passed as first-class-citizens to other network processes.
2. Lend a framework to mathematically verify the behavior of reflective, communicating processes and fundamentally concurrent systems of dynamic network topology.
3. Denote a fully scalable design which naturally accommodates industry trends in structural pattern matching, process continuation, Reactive API's, parallelism, asynchronicity, and behavioral types.

2.7.4 RhoLang - A Concurrent Language

Rholang is a fully featured, general purpose, Turing-complete programming language built from the rho-calculus. It is a behaviorally typed, **r**-eflective, **h**-igher **o**-rder process language and the official smart contracting language of RChain. Its purpose is to concretize fine-grained, programmatic concurrency.

Necessarily, the language is concurrency-oriented, with a focus on message-passing through input-guarded channels. Channels are statically typed and can be used as single message-pipes, streams, or data stores. Similar to typed functional languages, Rholang will support immutable data structures.

To get a taste of Rholang, here's a contract named `Cell` that holds a value and allows clients to get and set it:

```
contract Cell( get, set, state ) = {
  select {
    case rtn <- get; v <- state => {
      rtn!( *v ) | state!( *v ) | Cell( get, set, state )
    }

    case newValue <- set; v <- state => {
      state!( *newValue ) | Cell( get, set, state )
    }
  }
}
```

This contract takes a channel for `get` requests, a channel for `set` requests, and a `state` channel where we will hold a data resource. It waits on the `get` and `set` channels for client requests. Client requests are pattern matched via `case` classes[6].

Upon receiving a request, the contract joins `;` an incoming client with a request against the `state` channel. This join does two things. Firstly, it removes the internal `state` from access while this, in turn, sequentializes `get` and `set` actions, so that they are always operating against a single consistent copy of the resource - simultaneously providing a data resource synchronization mechanism and a memory of accesses and updates against the `state`.

In the case of `get`, a request comes in with a `rtn` address where the value, `v`, in `state` will be sent. Since `v` has been taken from the `state` channel, it is put back, and the `Cell` behavior is recursively invoked.

In the case of `set`, a request comes in with a `newValue`, which is published to the `state` channel (the old value having been stolen by the join). Meanwhile, the `Cell` behavior is recursively invoked.

[6] See Scala Documentation: Case Classes

Confirmed by `select`, only one of the threads in `Cell` can respond to the client request. It's a race, and the losing thread, be it getter or setter, is killed. This way, when the recursive invocation of `Cell` is called, the losing thread is not hanging around, yet the new `Cell` process is still able to respond to either type of client request.

For a more complete historical narrative leading up to Rholang, see Mobile Process Calculi for Programming the Blockchain.

2.8 Namespace Logic

For a blockchain solution of internet scale to be realizable, it, like the internet, must possess a logic to reason about the "location" of a resource. Specifically, how do we reference a resource? How do we determine which agents can access that resource under what conditions? In contrast to many other blockchains, where addresses are flat public keys (or hashes thereof), RChain's virtual address space will be partitioned into namespaces. **In a very general explanation, a namespace is a set of named channels.** Because channels are quite often implemented as data stores, a namespace is equivalently a set of contentious resources.

Q & A: Namespaces and double-spending

Q: Suppose Alice, Bob, and Carol are each in distinct namespaces and we have two payments: Alice-to-Bob and Alice-to-Carol. If I'm a node that only wants to care about Alice, how can I know that Alice doesn't double-spend?

A: A namespace is just a collection of names. All blockchain addresses are names. A collection can be described in a few ways. One of them is extensionally by explicitly spelling out each item in the collection. Another way is intensionally by providing a rule or program that either generates the collection or recognizes when an item is in the collection or out of the collection. The more interesting namespaces are the intensionally specified ones.

Now, the next step is to relate those to users, contracts, and nodes. Both users and contracts interact with each other via names. Nodes verify transactions in given namespaces, and transactions are i/o events across names (which are used as channels). Any transaction that involves two separate namespaces must be served by a collection of nodes that handles those namespaces. If there are no nodes that handle transactions that combine the namespaces, then the transaction cannot happen.

If there are nodes that combine the namespaces, then the consensus algorithm guarantees that all the nodes agree on the transactions. More specifically, they agree on the winners of every race. Thus, there can never be double spend. The biggest threat is to find composite namespaces that is served by few validators. Fortunately, you can see the validator power behind a namespace and decide whether to trust that namespace.

We have established that two processes must share a named channel to communicate, but what if multiple processes share the same channel? Transactional nondeterminism is introduced under two general conditions which render a resource contentious and susceptible to race conditions:

```
for(ptrn <- x){P1} | x!(@Q) | for(ptrn <- x){P2}
```

The first race condition occurs when multiple clients in parallel composition compete to *receive* a data resource on a named channel. In this case `P1` and `P2` , are waiting, on the named channel `x`, for the resource `@Q` being sent on `x` by another process. The clients will execute their continuations if and only if the correct value is witnessed at that location. In other cases where many clients are competing, many reductions may be possible, but, in this case, only one of two may result. One where `P1` receives `@Q` first and one where `P2` receives `@Q` first, both of which may return different results when `@Q` is substituted into their respective protocol bodies.

```
x!(@Q1) | for(ptrn <- x){P} | x!(@Q2)
```

The second race condition occurs when two clients compete to *send* a data resource on a named channel. In this case, two clients are each competing to send a data resource `@Q` to the client at the named channel `x`, but only one of two

transactions may occur - one where the receiving client receives `@Q1` first and one where it receives `@Q2` first, both of which may return different results when substituted into the protocol body of `P`.

For protocols which compete for resources, this level of nondeterminism is unavoidable. Later, in the section on consensus, we will describe how the consensus algorithm maintains replicated state by converging on one of the many possible transaction occurrences in a nondeterministic process. For now, observe how simply redefining a name constrains reduction in the first race condition:

```
for(ptrn <- x){P1} | x!(@Q) | for(ptrn <- v){P2} → P1{ @Q/ptrn } | for(ptrn <- v){P2}
```

–and the second race condition:

```
x!(@Q1) | for(ptrn <- x){P} | u!(@Q2) → P{ @Q1/ptrn } | u!(@Q2)
```

In both cases, the channel, and the data resource being communicated, is no longer contentious simply because they are now communicating over two distinct, named channels. In other words, they are in separate namespaces. Additionally, names are provably unguessable, so they can only be acquired when a discretionary external process gives them. Because a name is unguessable, a resource is only visible to the processes/contracts that have knowledge of that name[5]. Hence, sets of processes that execute over non-conflicting sets of named channels i.e sets of transactions that execute in separate namespaces, may execute in parallel, as demonstrated below:

```
  for(ptrn1 <- x1){P1} | x1!(@Q1) | ... | for(ptrnn <- xn){Pn} | xn!(@Qn) → P1{ @Q1/
↪ptrn1} | ... | Pn{ @Qn/ptrnn }

| for(ptrn1 <- v1){P1} | v1!(@Q1) | ... | for(ptrnn <- vn){Pn} | vn!(@Q1) → P1{ @Q1/
↪ptrn1} | ... | Pn{ @Qn/ptrnn }
```

The set of transactions executing in parallel in the namespace `x`, and the set of transactions executing in the namespace `v`, are double-blind; they are anonymous to each other unless introduced by an auxillary process. Both sets of transactions are communicating the same resource, `@Q`, and even requiring that `@Q` meets the same `ptrn`, yet no race conditions arise because each output has a single input counter-part, and the transactions occur in separate namespaces. This approach to isolating sets of process/contract interactions essentially partitions RChain's address space into many independent transactional environments, each of which are internally concurrent and may execute in parallel with one another.

Still, in this representation, the fact remains that resources are visible to processes/contracts which know the name of a channel and satisfy a pattern match. After partitioning the address space into a multiplex of isolated transactional environments, how do we further refine the type of process/contract that can interact with a resource in a similar environment? – under what conditions, and to what extent, may it do so? For that we turn to definitions.

2.8.1 Namespace Definitions

A namespace definition is a formulaic description of the minimum conditions required for a process/contract to function in a namespace. In point of fact, the consistency of a namespace is immediately and exclusively dependent on how that space defines a name, which may vary greatly depending on the intended function of the contracts the namespace definition describes.

A name satisfies a definition, or it does not; it functions, or it does not. The following namespace definition is implemented as an 'if conditional' in the interaction which depicts a set of processes sending a set of contracts to a set of named addresses that comprise a namespace:

1. A set of contracts, `contract1...contractn` , are sent to the set of channels (namespace) `address1...addressn`.
2. In parallel, a process listens for input on every channel in the `address` namespace.

[5] Namespace Logic - A Logic for a Reflective Higher-Order Calculus.

3. When a contract is received on any one of the channels, it is supplied to `if cond.`, which checks the namespace origin, the address of sender, the behavior of the contract, the structure of the contract, as well as the size of data the contract carries.
4. If those properties are consistent with those denoted by the `address` namespace definition, continuation `P` is executed with `contract` as its argument.

A namespace definition effectively bounds the types of interactions that may occur in a namespace - with every contract existing in the space demonstrating a common and predictable behavior. That is, the state alterations invoked by a contract residing in a namespace are necessarily authorized, defined, and correct for that namespace. This design choice makes fast datalog-style queries against namespaces very convenient and exceedingly useful.

A namespace definition may control the interactions that occur in the space, for example, by specifying:

- Accepted Addresses
- Accepted Namespaces
- Accepted Behavioral Types
- Max/Min Data Size
- I/O Structure

A definition may, and often will, specify a set of accepted namespaces and addresses which can communicate with the agents it defines.

Note the check against behavioral types in the graphic above. This exists to ensure that the sequence of operations expressed by the contract is consistent with the safety specification of the namespace. Behavioral type checks may evaluate properties of liveness, termination, deadlock freedom, and resource synchronization - all properties which ensure maximally "safe" state alterations of the resources within the namespace. Because behavioral types denote operational sequencing, the behavioral type criteria may specify post-conditions of the contract, which may, in turn, satisfy the preconditions of a subsequent namespace. As a result, the namespace framework supports the safe composition, or "chaining" together, of transactional environments.

2.8.2 Composable Namespaces - Resource Addressing

Until this point, we've described named channels as flat, atomic entities of arbitrary breadth. With reflection, and internal structure on named channels, we achieve depth.

A namespace can be thought of as a URI (Uniform Resource Identifier), while the address of a resource can be thought of as a URL (Uniform Resource Locator). The path component of the URL, `scheme://a/b/c`, for example, may be viewed as equivalent to an RChain address. That is, a series of nested channels that each take messages, with the named channel, `a`, being the "top" channel.

Observe, however, that URL paths do not always compose. Take `scheme://a/b/c` and `scheme://a/b/d`. In a traditional URL scheme, the two do not compose to yield a path. However, every flat path is automatically a tree path, and, as trees, these *do* compose to yield a new tree `scheme://a/b/c+d`. Therefore, trees afford a composable model for resource addressing.

Above, unification works as a natural algorithm for matching and decomposing trees, and unification-based matching and decomposition provides the basis of query. To explore this claim let us rewrite our path/tree syntax in this form:

```
scheme://a/b/c+d  s: a(b(c,d))
```

Then adapt syntax to the I/O actions of the rho-calculus:

```
s!( a(b(c,d)) )

for( a(b(c,d)) <- s; if cond ){ P }
```

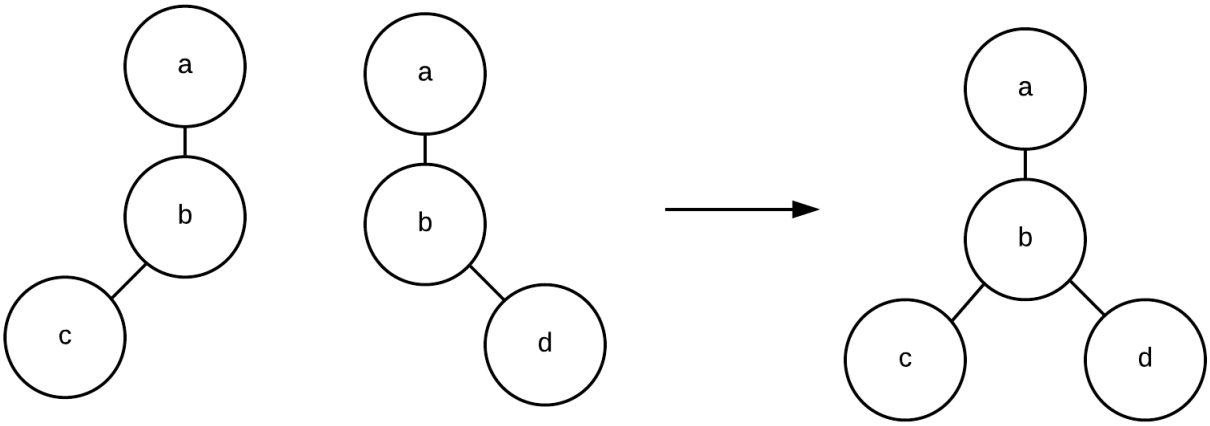


Fig. 4: Figure - Composable Tree Paths

The top expression denotes output - place the resource address `a(b(c,d)` at the named channel `s`. The bottom expression denotes input. For the pattern that matches the form `a(b(c,d))`, coming in on channel `s`, if some precondition is met, execute continuation `P`, with the address `a(b(c,d)` as an argument. Of course, this expression implicates `s`, as a named channel. So the adapted channel structure is represented:

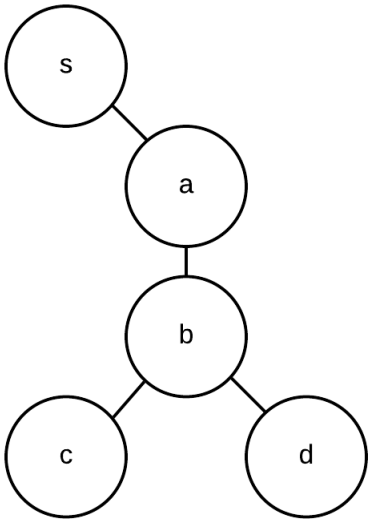


Fig. 5: Figure - URL Scheme as Nested Channels in Tree Structure

Given an existing address structure, and namespace access, a client may query for and send to names within that address structure. For example, when the rho-calculus I/O processes are placed in concurrent execution, the following expression denotes a function that places the quoted processes, `(@Q,@R)` at the location, `a(b(c,d))`:

```
for( a(b(c,d)) <- s; if cond ){ P } | s!( a(b(@Q,@R)) )
```

The evaluation step is written symbolically:

```
for( a(b(c,d)) <- s; if cond ){ P } | s!( a(b(@Q,@R)) ) → P{ @Q := c, @R := d }
```

That is, `P` is executed in an environment in which `c` is substituted for `@Q`, and `d` is substituted for `@R`. The updated tree structure is represented as follows:

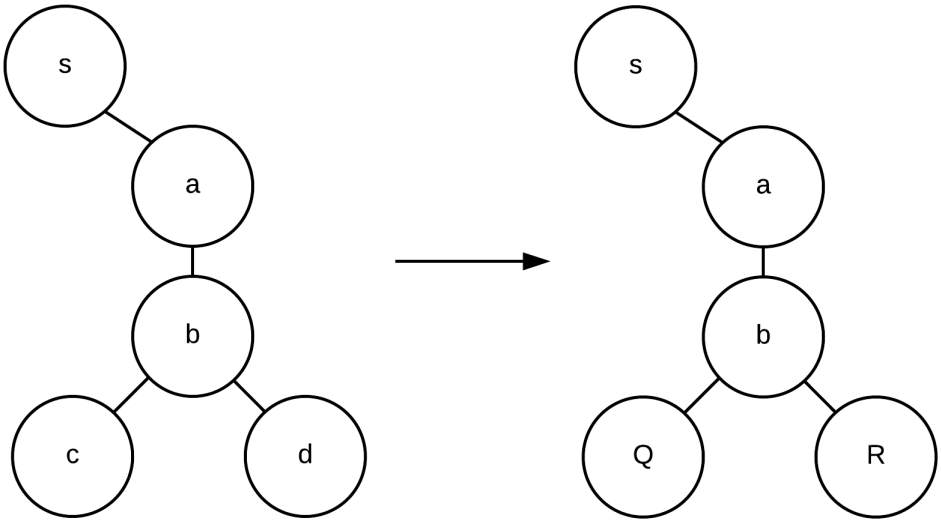


Fig. 6: Figure - Placing Processes at Channels

In addition to a flat set of channels e.g `s1...sn` qualifying as a namespace, every channel with internal structure is, in itself, a namespace. Therefore, `s`, `a`, and `b` may incrementally impose individual namespace definitions analogous to those given by a flat namespace. In practice, the internal structure of a named channel is an n-ary tree of arbitrary depth and complexity where the "top" channel, in this case `s`, is but one of many possible names in `s1...sn` that possess internal structure.

This resource addressing framework represents a step-by-step adaptation to what is the most widely used internet addressing standard in history. RChain achieves the compositional address space necessary for private, public, and consortium visibility by way of namespaces, but the obvious use-case addresses scalability. Not by chance, and not surprisingly, namespaces also offer a framework for RChain's sharding solution.

2.9 Execution Model

2.9.1 Overview

Each instance of the **Rho Virtual Machine** (RhoVM) maintains an environment that repeatedly applies the low-level rho-calculus reduction rule, expressed in the high-level Rholang contracting language, to the elements of a persistent key-value data store[1]. The "state" of RhoVM is analogous to the state of the blockchain.

The execution of a contract affects the *environment* and *state* of an instance of RhoVM. In this case, the usage of the term "environment" does not refer to the execution environment exclusively, but to the configuration of the key-value structure. Environment and state are the mapping of names to locations in memory, and of locations in memory to

[1] The RhoVM "Execution Environment" will later be introduced as the "Rosette VM". The choice to use Rosette VM hinged on two factors. First, the Rosette system has been in commercial production for over 20 years. Second, Rosette VM's memory model, model of computation, and runtime systems provide the support for concurrency that RhoVM requires. RChain has pledged to perform a modernized re-implementation of Rosette VM, in Scala, to serve as the initial RhoVM execution environment.

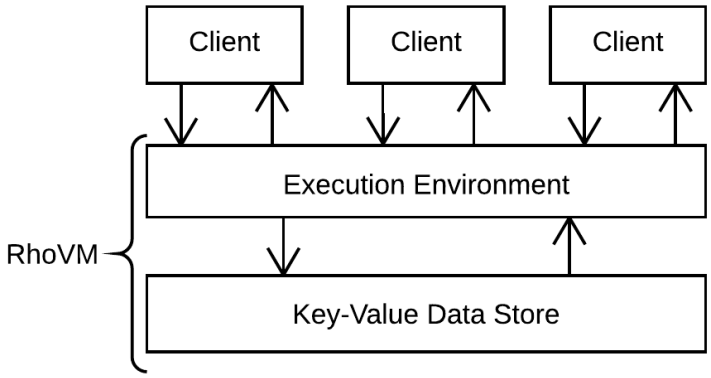


Fig. 7: *Figure - RhoVM as a back-to-back key-value store and execution engine*

values, respectively. Variables directly reference locations, which means that environment is equivalently a mapping of names to variables. A program typically modifies one or both of these associations at runtime. Environmental modifications occur with the lexical scoping rules of Rholang, and values may be simple or complex i.e. primitive values or complete programs.

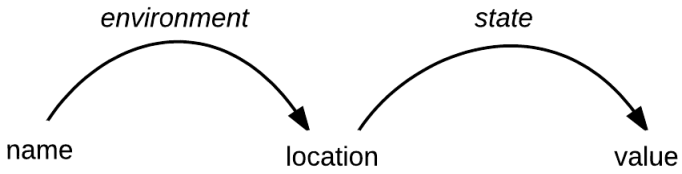


Fig. 8: *Figure - Two-stage binding from names to values*

RhoVM operates against a key-value data store. **A state change of RhoVM is realized by an operation that modifies which key maps to what value.** Since, like Rholang, RhoVM is derived from the rho-calculus model of computation, that operation is the low-level rho-calculus reduction rule. Effectively, the reduction rule, known as the "COMM" rule, is a substitution that defines a computation `P` to be performed if a new value is observed at a key. A key is analogous to a name in that it references a location in memory which holds the value being substituted. In the following example, `key` is a key and `val` is the value being substituted:

```
for ( val <- key )P | key! ( @Q ) -> P { @Q/val }
```

Barring consensus, this is the computational model of a concurrent protocol that stores a contract on the blockchain. On some thread, the output process `key!` stores the code of a contract `@Q` at the location denoted by `key`. On another thread running concurrently, the input process `for ( val <- key )P` waits for a new value `val` to appear at `key`. When some `val` appears at `key`, in this case `@Q`, `P` is executed in an environment where `@Q` is substituted for every occurrance of `val`. This operation modifies the value that `key` references i.e. `key` previously mapped to a generic `val` but now it maps to the code of a contract `@Q`, which qualifies a reduction as a state transition of the RhoVM.

The synchronization of an input and output process at `key` is the event that triggers a state transition of RhoVM. At first glance, the output process, which stores the contract `@Q` to the location denoted by `key`, appears to constitute a

state transition in itself. However, the rho-calculus reduction semantics have an *observability* requirement. For any future computation `P` to occur, the reduction rule requires that the input process `for ( val <- key) P` *observes* the assignment at `key`. This is because only the input term defines future computation, which means that the output term alone is computationally insignificant. Therefore, no *observable* state transition occurs until the input and output terms synchronize at `key`. This obvservability requirement is enforced at compile-time to prevent DDoS attacks by repeated output `key!(@Q)` invocation.

It has been demonstrated that an application of the rho-calculus reduction rule, to a data element of a key-value data store, constitutes a state transition of an instance of the RhoVM. The goal, however, is to verify and maintain every state transition that is specified by any contract to ever execute on an instance of the RhoVM. This means that the configuration history of the key-value data store must be maintained through modification, hence it being a *persistent* data structure. Therefore, each key must map to the verified history of reductions to occur at that location:

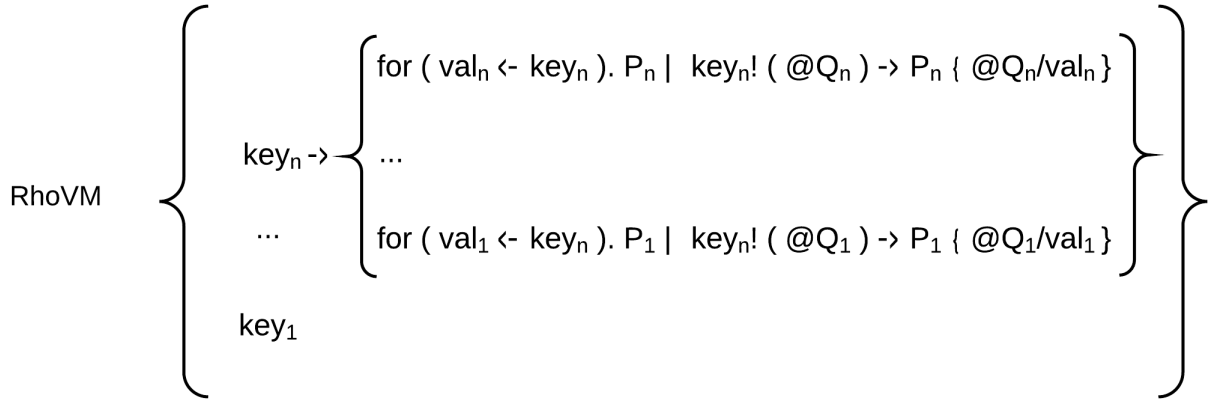


Fig. 9: *Figure - Reduction/transaction history of a location in memory*

Each key maps to a list of reductions which is, in fact, the "transaction history" of an address. The history of transactions `{ for(val1 <- keyn).P1 | keyn!(@Q1), ... , for(valn <- keyn).Pn | keyn!(@Qn) } -> { P1{@Q1/val1}, ... , Pn{@Qn/valn} }` denotes the modifications that have been made to the contract `@Q`, where `@Qn` is the most current version in store. It is important to recognize that this scheme is a top-level transaction on the RChain platform. The messages being passed are contracts themselves, which most often occurs in client-system, or system-system interactions. However, each contract `@Q` may, itself, execute many lower-level transactions on simpler values.

After a transaction/reduction is applied, it is subjected to consensus. Consensus verifies that the transaction history, `{ for(val1 <- keyn).P1 | keyn!(@Q1) ... for(valn <- keyn).Pn | keyn!(@Qn) }`, of `keyn`, is consistently replicated across all nodes running that instance of RhoVM. Once transaction histories are verified, the most recent transaction is added to the transaction history. The same consensus protocol is applied over the range of keys `{ key1 -> val1 ... keyn -> valn }` as transactions are committed to those locations.

By extension, transaction blocks represent sets of reductions that have been applied to elements of the persistent key-value store, and transaction histories represent verifiable snapshots of the state configurations and transitions of an instance of the Rho Virtual Machine. Note that the consensus algorithm is applied if, and only if, node operators propose conflicting reduction histories.

To summarize:

1. RhoVM is the composition of the rho-calculus reduction semantics, expressed in Rholang, and a persistent key-value data store.

2. The rho-calculus reduction rule substitutes the value at a key for another value, where a named channel corresponds to a key, and values may be simple or complex.
3. Substitutions are transactions, which manifest as differences in the bytecode stored at a key. The accurate replication of those bytecode differences, across all nodes operating that instance of RhoVM, is verified via the consensus algorithm.

A Brief Aside on Scalability

From the perspective of a traditional software platform, the notion of "parallel" VM instances is redundant. It is assumed that VM instances operate independently of each other. Accordingly, there is no "global" RhoVM. Instead, there is a multiplex of independently operating RhoVM instances running on nodes across the network at any given moment - each executing and validating transactions for their associated shards, or as we have been referring to them, namespaces.

This design choice constitutes system-level concurrency on the RChain platform, where instruction-level concurrency is given by Rholang. Hence, when this publication refers to a single instance of RhoVM, it is assumed that there are a multiplex of RhoVM instances simultaneously executing a different set of contracts for a different namespace.

2.9.2 Execution Environment

What Is Rosette?

Rosette is a reflective, object-oriented language that achieves concurrency via actor semantics. The Rosette system (including the Rosette virtual machine) has been in commercial production since 1994 in Automated Teller Machines. Because of Rosette's demonstrated reliability, RChain Cooperative has committed to completing a clean-room reimplementation of Rosette VM in Scala (targeting the JVM). There are two main benefits of doing so. First, the Rosette language satisfies the instruction-level concurrency semantics expressed in Rholang. Second, Rosette VM was intentionally designed to support a multi-computer (distributed) system operating on an arbitrary number of processors. For more information, see Mobile Process Calculi for Programming the Blockchain.

Model Checking and Theorem Proving

In the RhoVM and potentially in upstream contracting languages, there are a variety of techniques and checks that will be applied during compile-time and runtime. These help address requirements such as how a developer and the system itself can know a priori that contracts that are well-typed will terminate. Formal verification will assure end-to-end correctness via model checking (such as in SLMC) and theorem proving (such as in Pro Verif). Additionally, these same checks can be applied during runtime as newly proposed assemblies of contracts are evaluated.

Discovery Service

An advanced discovery feature that will ultimately be implemented enables searching for compatible contracts and assembling a new composite contract from other contracts. With the formal verification techniques, the author of the new contract can be guaranteed that when working contracts are plugged together they will work as well as a single contract.

2.9.3 Compilation

To allow clients to execute contracts on the RhoVM, RChain has developed a compiler pipeline that starts with Rholang source-code. Rholang source-code first undergoes transcompilation into Rosette source-code. After analysis, the Rosette source-code is compiled into a Rosette intermediate representation (IRs), which undergoes optimization. From

the Rosette IR, Rosette bytecode is synthesized and passed to the VM for local execution. Each translation step within the compilation pipeline is either provably correct, commercially tested in production systems, or both. This pipeline is illustrated in the figure below:

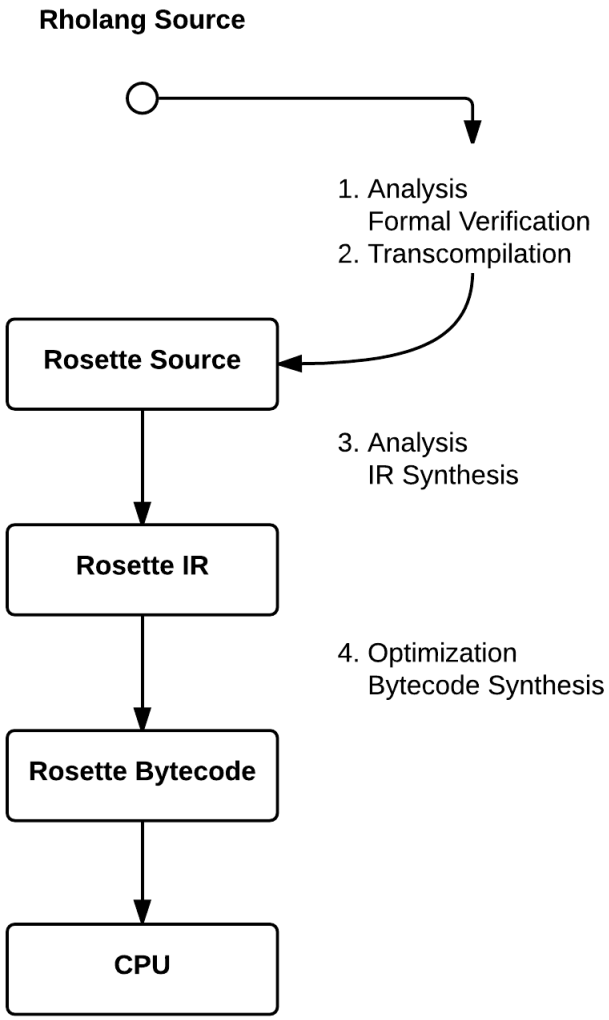


Fig. 10: *Figure - RChain compilation strategy*

1. **Analysis**: From Rholang source-code, or from another smart contract language that compiles to Rholang, this step includes:
   (a) analysis of computational complexity
   (b) injection of code for the rate-limiting mechanism
   (c) formal verification of transaction semantics
   (d) desugaring of syntax
   (e) simplification of functional equivalencies

2. **Transcompilation**: From Rholang source-code, the compiler:
   (a) performs a source-to-source translation from Rholang to Rosette source-code.
3. **Analysis**: From Rosette source-code, the compiler performs:
   (a) lexical, syntactic, and semantic analysis of the Rosette syntax, construction of the AST; and
   (b) synthesizes a Rosette intermediate representation
4. **Optimization**: From Rosette IR, the compiler:
   (a) optimizes the IR via redundancy elimination, subexpression elimination, dead-code elimination, constant folding, induction variable identification and strength simplification
   (b) synthesizes bytecode to be executed by the Rosette VM

Rate-limiting Mechanism

The compilation pipeline will implement a rate-limiting mechanism that is related to some calculation of processing, memory, storage, and bandwidth resources. Because the rho-calculus reduction rule is the atomic unit of computation on the RChain platform, the calculation of computation complexity is necessarily correlated to the amount of reductions performed per contract. This mechanism is needed in order to recover costs for the hardware and related operations. Although Ethereum (Gas) has similar needs, the mechanisms are different. Specifically, the metering will not be done at the VM level, but will be injected in the contract code during the analysis phase of compilation. For more details visit the **'developer wiki'_**. Compiler work can be seen on GitHub.

2.10 Storage and Query

2.10.1 Overview

The Storage and Query network layer *appears* to each node as a local, asynchronously accessed database with rented storage. Behind the scenes, however, the Storage and Query layer is fully decentralized and subject to the consensus algorithm. In accordance with the micro-transaction capabilities inherent to blockchain solutions, dApp users on RChain pay for resources (compute, memory, storage, and network) using tokens. The RChain design considers all storage "conserved", although not all data will be conserved forever. Instead, data storage will be rented and will cost producers of that data in proportion to its size, complexity, and lease duration. Consumers may also be required to pay for retrieval access. Data producers and consumers indirectly pay node operators.

The simple economic reason justifying leasing is that storage must be paid by someone; otherwise it cannot be stored reliably or "forever". We've chosen to make the economic mechanism direct. It is an environmentally unfriendly idea that storage is made "free" only to subsidize it by an unrelated process. A small part of the real cost is measurable in the heat signatures of the data centers that are growing to staggering size. This charging for data as it is accessed also helps reduce "attack" storage i.e the storage of illegal content to discredit the technology.

A variety of data is supported, including public unencrypted json, encrypted BLOBs, or a mix. This data can also be simple pointers or content-hashes referencing off-platform data stored in private, public, or consortium locations and formats.

2.10.2 Data Semantics

The RChain blockchain will store the state, local transaction history, and the associated continuations of a contract. Like Ethereum, the RChain blockchain will also implement crypto-economically verifiable transactional semantics to create a linear temporal history of computation performed on the platform. Note that the math underlying this

blockchain semantic structure is known as a Traced Monoidal Category. For more detail see Masahito Hasegawa's paper on this topic, Recursion from Cyclic Sharing:Traced Monoidal Categories and Models of Cyclic Lambda Calculi.

Data Access Layer and Domain Specific Language

SpecialK is the DSL for data access, while KVDB is a distributed-memory data structure behind the DSL. SpecialK defines distributed data-access patterns in a consistent way, as shown below:

	Item-level read and write (distributed locking)	Database read and write	Publish / Subscribe messaging	Publish / Subscribe with history
Data	Ephemeral	Persistent	Ephemeral	Persistent
Continuation(K)[1]	Ephemeral	Ephemeral	Persistent	Persistent
Producer Verb[2]	Put	Store	Publish	Publish with History
Consumer Verb	Get	Read	Subscribe	Subscribe

Figure - SpecialK's Data Access Patterns

From the point of view of the SpecialK DSL and API, when it performs a data-access action, such as the verb Get (with a pattern), it is indifferent to whether that data it is stored locally or remotely i.e on some other network node. There is a single query mechanism regardless.

The 2016 and prior SpecialK technology stack (Agent Services, SpecialK, and KVDB, with RabbitMQ and MongoDB) delivered a decentralized Content Delivery Network, although it was neither metered nor monetized. The SpecialK & KVDB components sit on top of MongoDB and an Advanced Message Queuing Protocol (ZeroMQ is being explored) to create the decentralized logic for storing and retrieving content, both locally and remotely. The current 1.0 implementations of SpecialK and KVDB are written in Scala and are in GitHub.

The query semantics vary depending on which level in the architecture is involved. At the SpecialK level, keys are prolog expressions, which are later queried via datalog expressions. Higher up in the architecture, prolog expressions of labels are used for storage, and datalog expressions of labels are used for query. In RChain, the SpecialK and KVDB layers will be reimplemented in Rholang (versus the prior implementation in Scala with custom implementation of delimited continuations and code serialization).

For more information, see SpecialK & KVDB – A Pattern Language for the Web.

KVDB - Data & Continuation Access, Cache

Data will be accessed using the SpecialK semantics, while physically being stored in a decentralized, Key-Value Database known as "KVDB". A view of how two nodes collaborate to respond to a get request is shown below:

1. The node first queries its in-memory cache for the requested data. Then if it is not found it,
2. queries its local store, and, if it is not found, stores a delimited continuation at that location, and
3. queries the network. If and when the network returns the appropriate data, the delimited continuation is brought back in scope with the retrieved data as its parameter.

[1] Note that by convention a continuation function is represented as a parameter named k.

[2] This is only a subset of the verbs possible under this decomposition of the functionality. The verb fetch, for example, gets the data without leaving a continuation around, if there is no data available.

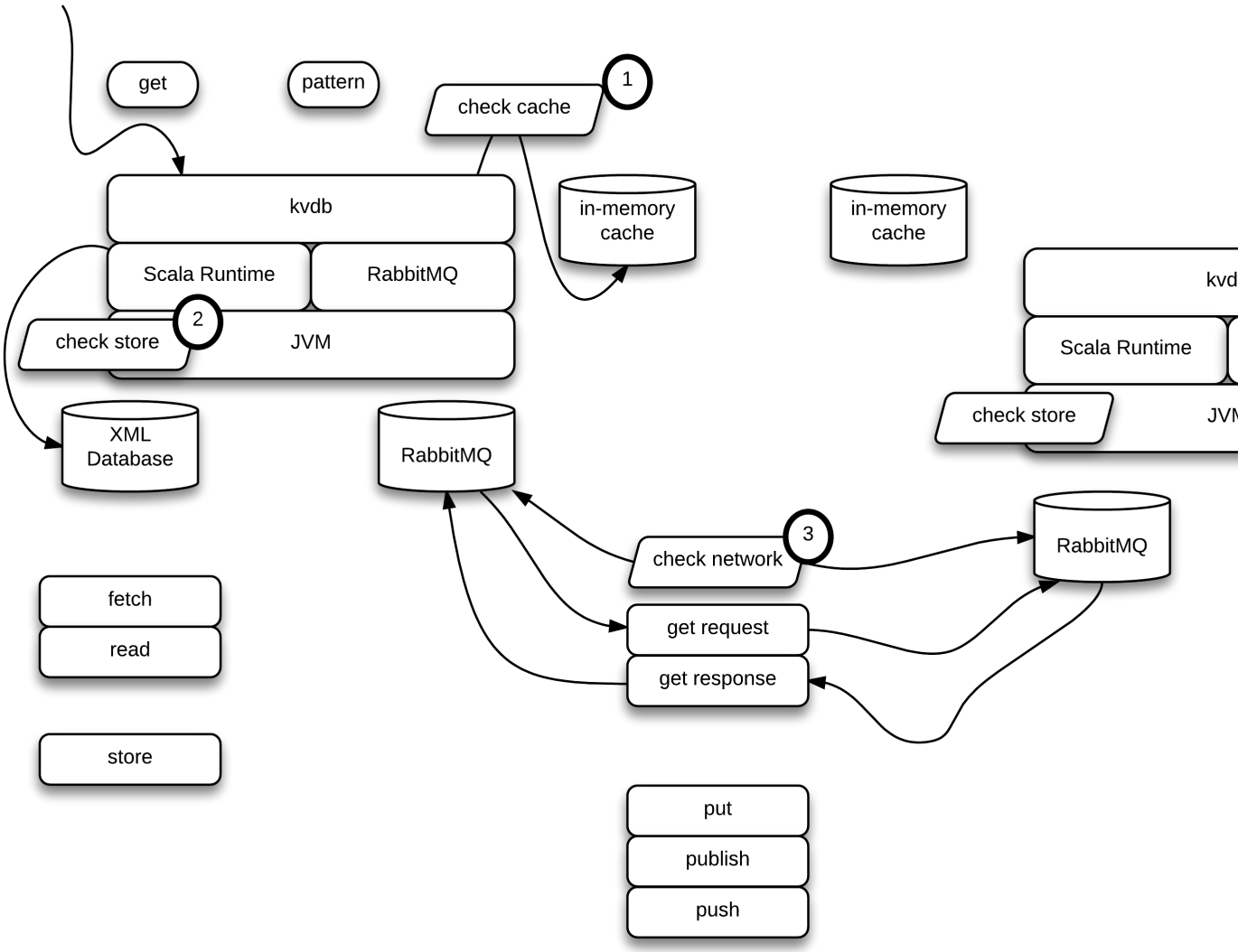


Fig. 11: *Figure - Decentralized Data Access in SpecialK*

Why did RChain not select IPFS (InterPlanetary File System) for distributed memory? In addition to carrying centralization risks, IPFS uses a path to get to content, whereas SpecialK uses entire trees (and trees with holes in them) to get to content. IPFS has an intuitive path model, but that design begs the question on how to do queries. SpecialK started from the query side of addressing. Now, the RChain project can benefit from the IPFS work, including their hashing for addressing content, once the SpecialK query semantics are in place. SpecialK can also utilize a randomly generated flat key that has no correlation to the data.

2.10.3 P2P Node Communications

The SpecialK decentralized storage semantics necessitate a node communications infrastructure. Similar to other decentralized implementations, the P2P communications component handles node discovery, inter-node trust, and communication. The current implementation uses RabbitMQ, although ZeroMQ is being considered.

2.11 Casper Consensus Algorithm

Nodes that take on the validation role have the function to achieve consensus on the blockchain state. Validators also assure a blockchain is self-consistent and hasn't been tampered with and protect against Sybil attack.

The Casper consensus protocol includes stake-based bonding, unbonding, and betting cycles that result in consensus. The purpose of a decentralized consensus protocol is to assure consistency of blockchains or partial blockchains (based on namespaces), across multiple nodes. To achieve this any consensus protocol should produce an outcome that is a proof of the safety and termination properties of class of consensus protocols, under a wide class of fault and network conditions.

RChain's consensus protocol uses stake-based betting, similar to Ethereum's Casper design. This is called a "proof-of-stake" protocol by the broader blockchain community, but that label leads to some misperceptions including overstated centralization risks. Validators are bonded with a stake, which is a security deposit placed in an escrow-like contract. Unlike Ethereum's betting on a whole blocks, RChain's betting is on logical propositions. A proposition is a set of statements about the blockchain, for example: which transactions (i.e. proposed state transitions) must be included, in which order, which transactions should not be included, or other properties. A concrete example of a proposition is: "transaction t should occur before transaction s" and "transaction r should not be included". For more information, see the draft specification Logic for Betting – On Betting on Propositions.

At certain rendezvous points validators compute a maximally consistent subset of propositions. In some cases, this can be computationally hard and take a long time. Because of this a time-out will exist, which, if reached forces validators to submit smaller propositions. Once there is consensus among the validators on the maximally consistent subset of propositions, the next block can easily be materialized by finding a minimal model under which the propositions are valid. Because of this design and because of the transactional isolation per namespace, most blocks can be synthesized in parallel.

Let's walk through the typical sequence:

1. A validator is a node role. Validators each put up a stake, which is akin to a bond, in order to assure the other validators that they will be good actors. The stake is at risk if they aren't a good actor.
2. Clients send transaction requests to validators.
3. Receiving validators then create a proposition including a recent transaction.

Note: consensus is executed only when transactional history is inconsistent between nodes

4. There are sets of betting cycles among nodes:
    (a) The originating validator prepares a bet, which includes the following:
    - *source* = the origin of the bet

- *target* = the destination or target for the bet
- *claim* = the claim of the bet. This is a block, a proposition, or maximally consistent subset of propositions
- *belief* = the player's confidence in the claim given the evidence in the justification. This is a denotation of the betting strategy used by the validator.
- justification. This is evidence for why it is a reasonable bet.

(b) The validator places the bet.

(c) The receiving validator evaluates the bet. Note, these justification structures can be used to determine various properties of the network. For example, an algorithm can detect equivocation, or create a justification graph, or detect when too much information is in the bet. Note how attack vectors are considered, and how game theory discipline has been applied to the protocol design.

5. The betting cycles continue working toward a proof. Note:

(d) The goal of the betting cycle is for the validator nodes to reach consensus on a maximally consistent set of propositions.

(e) A prerequisite condition for the proof is that of the validators are behaving in a reasonable fashion.

(f) Eventually the betting cycle will and must converge.

(g) The processing is partially synchronous during convergence.

(h) With by-proposition betting, the design will be able to synthesize much bigger chunks of the blockchain all at once. Cycles can converge quickly when there are no conflicts. The point of the by-proposition approach is that several blocks can be materialized all at once. This proposal gets around block size limits. There's no argument about it because the maximal consistent set of propositions might allow for hundreds or even thousands of blocks to be agreed all at once. This will create a huge speed advantage over existing blockchains.

(i) For each betting cycle a given validator node may win or lose their bet amount.

6. Scalability is achieved via a fine-grained partitioning of proposals and via nesting (recursion) of the consensus protocol. Blocks are synthesized by the protocol when there is agreement on the set of maximally-consistent propositions, and this occurs when there is a proof of convergence among the bets. The current betting cycle then collapses.

For additional information, see:

- Consensus Games: An Axiomatic Framework for Analyzing and Comparing a Wide Range of Consensus Protocols.
- For more detail on RChain's consensus protocol, see Logic for Betting – On Betting on Propositions.
- To find out more about Ethereum's Casper and discussions in the Ethereum Research Gitter and Reddit/ethereum.
- The math underlying the betting cycle is an Iterated Function System. Convergence corresponds to having attractors (fix-points) to IFS. With this, we can prove things about convergence with awards and punishments. We can give validator-node-betters maximum freedom. The only ones that are left standing are validators that are engaged in convergent betting behavior.

2.12 Applications

Any number and variety of applications can be built on top of the RChain Platform that provide a decentralized public compute utility. These include, for example:

- Wallets
- Exchanges
- Oracles & External Adapters
- Custom Protocols
- Smart Contracts
- Smart Properties
- DAOs
- Social Networks
- Marketplaces

2.12.1 Contract Development & Deployment

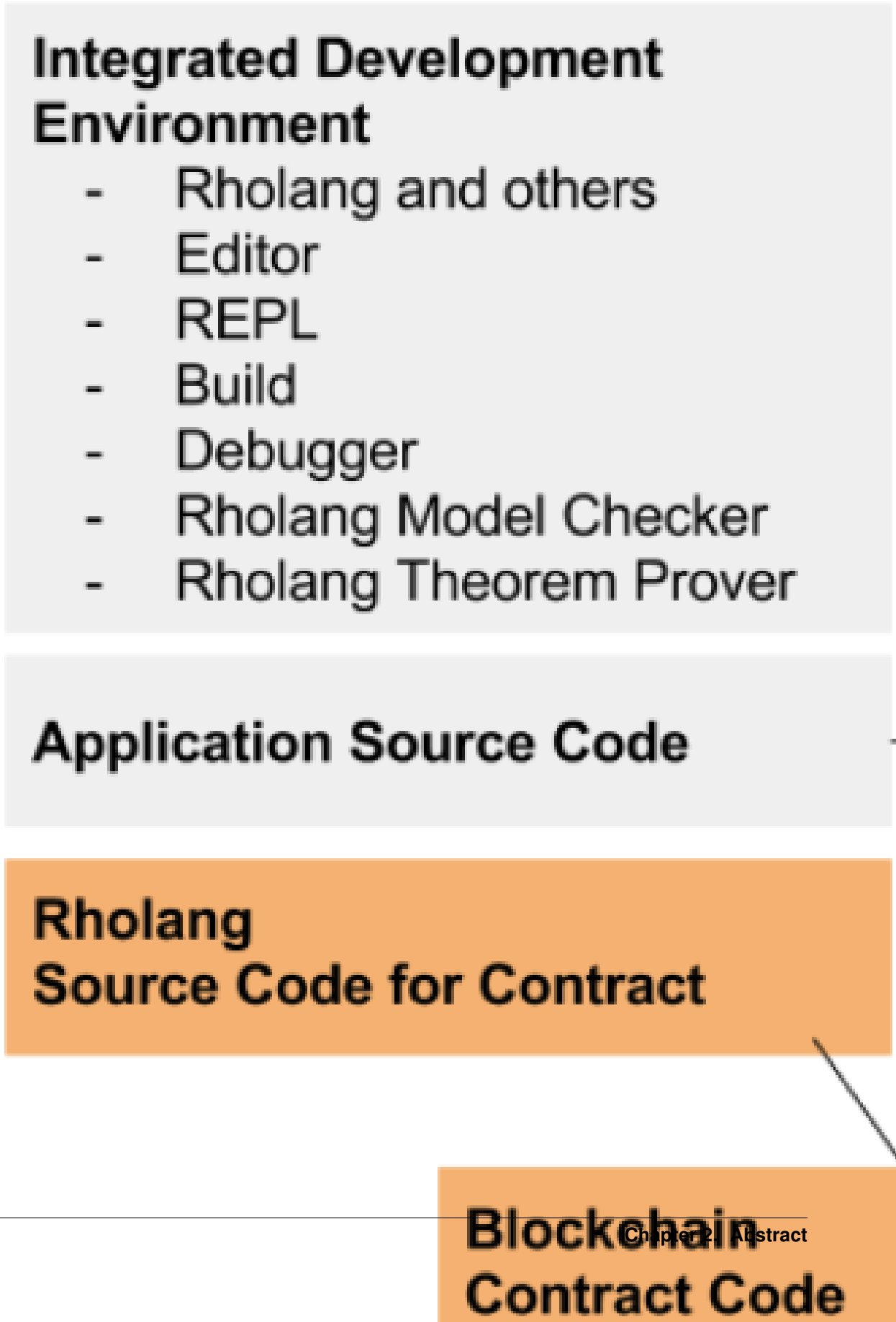
The purpose of this next discussion is to illustrate how namespaces allow for heterogeneous deployment of contracts and contract state. Namespaces is one of the crucial features for sharding, and with that we get the benefits analogous of sidechains, private chains, consortium chains, as well as the distinction between test and production, all under one rubric.

For example, the following diagram depicts some of the possible development, test, and deployment configurations and considerations, and how release management is enabled using namespaces and sharding.

We'll collaborate with IDE tool vendors to integrate Rholang and validation tools.

2.13 References

References


Integrated Development Environment
- Rholang and others
- Editor
- REPL
- Build
- Debugger
- Rholang Model Checker
- Rholang Theorem Prover
Application Source Code
Rholang
Source Code for Contract
Blockchain
Contract Code

Bibliography

[MiBH] Miller, H., Burmako, E., Haller, P.: Reflection Overview, http://docs.scala-lang.org/overviews/reflection/overview.html.

[Scal] Scala: Sequence Comprehensions, http://docs.scala-lang.org/tutorials/tour/sequence-comprehensions.html.

[Scal] Scala: Continuations, http://www.scala-lang.org/files/archive/api/2.11.8/scala-continuations-library/?_ga=1.83694417.619951421.1484537916#scala.util.continuations.package.

[Scal] Scala: Case Classes, http://docs.scala-lang.org/tutorials/tour/case-classes.

[MeRa05] Meredith, L., Radestock, M.: A Reflective Higher-order Calculus. Electronic Notes in Theoretical Computer Science. 141, 49–67, http://www.sciencedirect.com/science/article/pii/S1571066105051893?via%3Dihub

[MeRa05] Meredith L., Radestock M: Namespace Logic - A Logic for a Reflective Higher-Order Calculus. Trustworthy Global Computing Lecture Notes in Computer Science 353–369. doi: 10.1007/11580850_19, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.9601