HotPocket

A Blockchain-Agnostic DApp Engine

by

Scott Chamberlain, Richard Holland, and Ravin Perera

14 June 2021

1. INTRODUCING HOTPOCKET

1.1 HotPocket is a UNL (Unique Node List) consensus engine that converts any number of Linux machines into a mini-blockchain capable of cheaply and speedily running any dApp in any language at scale. These dApps can interact with almost any blockchain, including the XRP Ledger.

The Promise of DApps

1.2 DApps - decentralised applications - are a new form of software that runs on blockchains without centralised ownership or control. DApps are more secure and reliable than traditional applications. Permissionless and censorship-resistant, dApps enable novel use-cases, like self-sovereign identity, improved privacy, decentralised finance, and tokenised art, rights, and assets.

The Challenge with Early DApp Platforms

1.3 Early dApp platforms have struggled to scale due to high fees, slow transaction times, and inflexibility as to language and functionality, all legacies of the blockchains on which they run. These drawbacks have hampered the mainstream adoption of dApps and the realisation of the benefits they offer.

1.4 These limitations are particularly acute if the dApp is supposed to automate legal obligations or relationships. Lack of scaling means "off-chain" computation or storage, giving an inevitable element of centralisation (or single point of failure) to what is otherwise supposed to be a decentralised system.

HotPocket's Solution

1.5 HotPocket overcomes these drawbacks so anyone can launch any dApp they desire. It has five main components:

(a) **UNL Consensus Protocol:** HotPocket is a Unique Node List (UNL) based consensus protocol that allows multiple Linux machines to become a mini-blockchain by enforcing consensus rules on inputs and outputs and maintaining a shared, canonical state.

(b) **Rapid State Sync:** A bundle of additional features designed to make it as easy as possible to spin up a HotPocket Node.

(c) **Contract Lifecycle Management:** HotPocket ensures all contracts have the same configuration and allows contracts to be upgraded automatically at consensus.

(d) **Daemon:** HotPocket clusters can be "nailed" to any layer 1 blockchain through a daemon, called "Sashimono", that co-ordinates the cluster through cryptographically secure messages posted to the layer 1 blockchain.

(e) **Minimal setup:** Hot Pocket enables new nodes to join an existing contract with minimal known information to sync the new node with the cluster.

1.6 In concert, these components provide an "out of the box" solution for a network of HotPocket Nodes running a mini-blockchain capable of supporting a wide variety of cheap, speedy dApps capable of interacting with almost any layer 1 chain.

In Production

1.7 HotPocket is in production. We have working code that has maintained consensus across a globally distributed cluster of ten Nodes. The final step is to create the coordination and configuration daemon, Sashimono, to allow the decentralised deployment and management of HotPocket clusters "in the wild" from any layer 1 infrastructure.

2. UNL-CONSENSUS PROTOCOL

2.1 The HotPocket Consensus Protocol uses a cooperative UNL-based consensus algorithm akin to the Ripple Consensus Protocol, abstracted for any transaction or input type.

Technical Specs

Language: C++17

Supported operating system: Linux (a planned container solution will allow any OS/platform)

Supported smart contract format: Any POSIX-compliant binary.

Minimum system requirements: 1 VCPU, 1GB RAM (Standard VM cost $5 USD per month)

Consensus round time: 2 seconds round time on globally distributed 10-node cluster (10 x $5 USD).

Smart contract I/O throughput: 100 tx/s on globally distributed 10-node cluster (10 x $5 USD).

Max smart contracts per-node: Theoretically limited by number of TCP ports. Practical limit not yet measured.

Max nodes per cluster: Not yet measured.

2.2 A HotPocket smart contract is a POSIX compliant executable that can receive user input, producing output, and maintaining persisted contract state (see Figure 1.) It can be developed as a POSIX-compliant executable using any language the developer chooses.

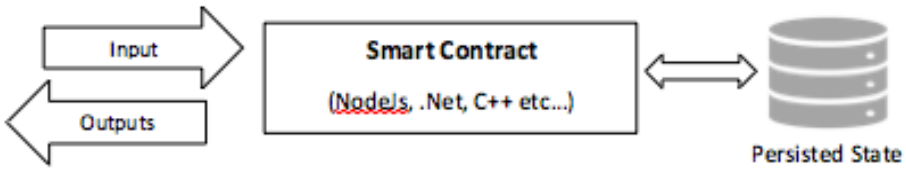

Figure 1—HotPocket Smart Contract Information Flow

2.3 Such a dApp can be hosted on multiple HotPocket nodes to create a cluster. HotPocket handles agreement on all inputs, outputs and state via consensus. (See Figure 2.)

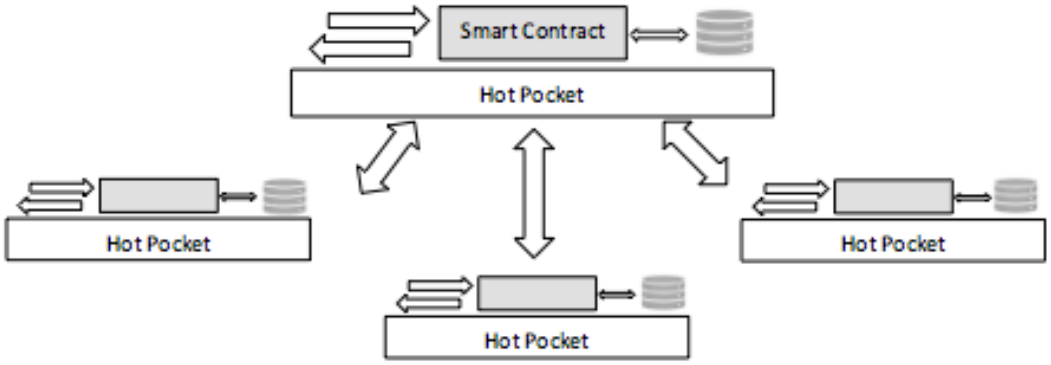

Figure 2—HotPocket Cluster Example Topology

2.4 HotPocket smart contract instances are made available as Docker containers that are runnable on a Linux host. All HotPocket Docker containers make use of a FUSE file system layer installed on the underlying Linux host. The sandbox isolation offered by HotPocket Docker containers offers a secure execution environment for each smart contract instance running on the same host. Smart contract developers can package the necessary execution dependencies into the Docker container and have them run on the swarm of hosting servers.

Consensus Engine

2.5 The HotPocket consensus engine works as follows:

(a) **Set Up:** The programmer configures a set of servers and builds a Unique Node List of these servers' public keys and a peer list of IPs and ports. This configuration is copied to all servers in the contract's network.

(b) **Synchronisation:** During execution, each HotPocket instance connects to its peers and synchronizes the current contract state across all UNL peers according to an 80% consensus. Once state transfer and synchronization are achieved, the contract's network collects inputs from the contract's users.

(c) **Inputs from Users:** Users can connect to any node not explicitly configured to reject their connections. Users identify themselves by proving ownership of a public key and their inputs are then circulated into the consensus mechanism, akin to transactions being circulated into the Bitcoin, XRPL, or Ethereum networks.

(d) **Consensus on Inputs:** The contract's nodes then execute a consensus round deciding which user inputs made it into this block and which will be held off for next block. Other essential consensus information, like the time of the round, the current contract state, and the identity of the last closed ledger (the canonical state of the network) also enter into consensus. After three rounds, consensus is reached by an 80% majority of UNL peers. Non-UNL peers can also observe consensus if the contract network is configured as a public network.

(e) **Execution**: Upon consensus, each HotPocket node simultaneously executes the smart contract binary and provides the same set of user inputs in the same canonical order to the binary. The smart contract processes the batch of user inputs and produces a set of contract outputs. These outputs are of two forms: updates to the contract's state and user outputs (to be sent back to users).

(f) **Node Party Line & Sub-Consensus Feature:** During smart contract execution, the UNL nodes may communicate with each other over a broadcast service provided by HotPocket. This is called the Node Party Line. This feature allows nodes to run sub-consensus agreement and information sharing before exiting and alleviates the need for each node to behave completely deterministically. For example, the nodes may wish to pass a multi-sig transaction between themselves and each sign with a key only that individual node possesses, before agreeing on a canonical final signed multi-sig transaction and exiting.

(g) **User Outputs:** Once the smart contract has executed, another consensus round takes place to ensure all nodes produced the same output and the same changes to their state. Once the result of the execution is agreed upon, the user outputs are passed to the users for whom they were destined according to the contract's internal programming or dropped if the user is no longer connected anywhere on the contract's network. In practice, this new consensus round also gathers up a new batch of user inputs to feed into the contract's next execution.

3. RAPID STATE SYNCING

3.1 To enable new nodes to quickly catch-up state, HotPocket uses a state management system modelled from BitTorrent.

3.2 Each contract execution allows the contract binary to read and write into its state folder which is in fact a mounted FUSE device managed by a HotPocket sub-process. Each time the contract writes to, or alters, its on-disk state the FUSE sub-process updates its Merkel tree representation of the contract's state. The deltas between Merkel trees are then computed to allow efficient transfer of only the changed data between two arbitrarily distant states in the ledger chain.

3.3 This means nodes do not need to replay old ledgers to catch up state, which would likely be otherwise impossible for high throughput contracts.

4. CONTRACT CONFIG SYNC

4.1 HotPocket has two features to help elegantly manage contract lifecycles.

Online Configuration

4.2 First, the contract config is subjected to consensus ensuring all contract nodes use the same configuration which effects deterministic execution. Contracts can also update their own configuration at runtime and rely on consensus to ensure all the nodes retain the same configuration.

Self-Editable Contracts

4.3 Second, HotPocket smart contracts can be configured (if desired) to "live" among the contract data, subjecting the contract binaries and upgrade activities to consensus.

4.4 HotPocket offers a handoff mechanism to perform contract upgrades in between consensus rounds by means of an installation shell script provided by the contract. The results of the upgrade are automatically validated in subsequent consensus rounds.

Auto-Lifecycle Management

4.5 Combined, these two contract management features mean HotPocket offers a rich administrative environment in which the contract can self-manage its own life cycle.

5. SASHIMONO - HOTPOCKET'S DAEMON

5.1 Coordinating the rollout of a Hot Pocket smart contract would ordinarily require manual server setup and configuration for each HotPocket node in the contract. This is cumbersome and centralising.

5.2 A better solution is to dedicate a selection of servers for the collective purpose of running logical nodes from various HotPocket contracts from time to time, and then coordinate these from a unified and decentralised command point.

5.3 Thus, we have proposed that HotPocket have a daemon, called "Sashimono", (after the Japanese woodworking technique of building without nails or visible joins) that runs on each server that will host HotPocket nodes in the user's network. The daemon:

(a) Listens to a "layer 1" message board of which all its companion Nodes are members.

(b) Sends cryptographically secure messages to companion Nodes and post them to on the "layer 1" message board.

(c) In conjunction with a (client-side) developer IDE, seamlessly coordinate the rollout and configuration of HotPocket instances across companion Nodes.

5.4 The daemon will be configured so that any "layer 1" blockchain could be the message board. That is, all Nodes would be listening to a decentralised point of truth that could be any layer 1 blockchain.

5.5 As an out-of-the-box functionality, the daemon would listen to an account on the XRPL or an XRPL Hook.

6. MINIMAL SETUP RULES

6.1 Hot Pocket enables new nodes to join an existing contract with minimal known information.

6.2 To join a contract, all a new node needs to know is:

(a) The Contract ID.

(b) One UNL key.

(c) One peer address.

6.3 With this information, HotPocket can sense the network and figure out the missing information (roundtime, contract configuration, UNL, peers etc…) to sync the new node with the cluster.

7. GREATER FLEXIBILITY, MORE USEFUL DAPPS

7.1 HotPocket dApps don't **run on** blockchains, they **are** blockchains. Each dApp is its own blockchain with its own chain history and dedicated nodes, making them incredibly flexible.

7.2 DApps may be public or private. DApps may call external services, read and write data directly to disk and the web, and generally perform any task a regular program can, without centralisation or trusted third parties and without requiring the programmer to implement their own consensus mechanisms.

7.3 This flexibility solves many problems that limit mass adoption of dApps including:

(a) **Privacy Compliance:** dApps can encrypt data, run only on hosts in a given jurisdiction, or only on hosts that have agreed to meet privacy regulations.

(b) **Scale & Flexibility:** dApps can run on as few or as many hosts as the dApp developer desires from a cost and security perspective.

(c) **On-Demand Oracles:** HotPocket dApps can elect a sub-set or jury of their own nodes to get data from off-chain, agree on the truth, and report to the rest of the chain as a bespoke, on-demand oracle; and

(d) **Enhanced Security:** dApps can detect when a host has become compromised or untrustworthy, shut down that instance of the dApp, and reload it on another, more trusted Node.

7.4 HotPocket's daemon can be configured to use almost any layer 1 blockchain as the coordinating message board, although its out-of-the-box configuration favours the XRPL because of that chain's speed and exceptionally low fees.

7.5 Thus, HotPocket is a "bolt-on" layer 2 smart contract solution that allows almost any layer 1 blockchain to run any dApp cheaply and speedily in any language at any scale, including the XRPL.

8. NEXT STEPS – PILOT DAPPS

8.1 As of May 2021, HotPocket is production ready. A demo of a global cluster of nodes can be viewed here.

8.2 We are now focused on building and testing the Sashimono daemon, with out-of-the-box support for XRP Ledger.

8.3 Shortly, we will present pilots of dApps as examples of the kinds of applications that can run on HotPocket:

(a) **iXRPL:** a Self-KYC identity solution running on a HotPocket smart contract that allows users to maintain custody and ownership of their verified identity docs and status using an NFT issued on the XRP Ledger.

(b) **Digital Cows:** a HotPocket smart contract for 24/7/365 trading of digital interests in Australian cattle.