# The *SEALD* Model

Ben Hardekopf, Divyakant Agrawal, Tevfik Bultan, Yufei Ding, Amr El Abbadi
Yu Feng, Chandra Krintz, Tim Sherwood, and Rich Wolski
Computer Science Department
Univ. of California, Santa Barbara
Technical Report No. 2020-01

March 1, 2020

We believe that a complete system for IoT must address the following challenges before it can become a disruptive innovation.

- **Programming Heterogeneity**: The Internet of Things will comprise an almost enumerable number of devices, processor architectures, storage components, and communication systems. It is infeasible to develop software for such a vast plethora of platforms without a unifying set of programming abstractions that can be implemented on all systems. Moreover these abstractions must be able to incorporate new hardware and devices as they emerge and facilitate optimization of scale, speed, storage, and energy use, among other metrics.

- **Program Correctness**: Because IoT software is often deployed in remote locations without the ability to modify or update it (e.g. on devices that do not support "over-the-radio" software ingress) ensuring the correctness of software *before* it is deployed is essential. Using a "rip-and-replace" strategy to debug faulty and insecure software is infeasible. The next generation of programming tool chains and runtime systems must concurrently facilitate rigorously proven correctness, correct-by-construction application development, and runtime enforcement if/when doing so statically is not possible.

- **Performance Optimization**: To facilitate scale and robustness, IoT applications must be optimized for both power consumption and performance and their deployments must be specialized (in both hardware and software) to facilitate highly efficient data analytics and machine learning across all tiers. Such optimization is only possible, we believe, via codesign of hardware, software, and system advances.

- **End-to-end Trustworthiness**: Unlike "the cloud" which is typically operated by a single vendor for most applications, IoT applications must be able to span trust domains. Enforcing trustworthiness end-to-end in IoT settings requires a combination of new secure hardware, program analyses and verification, security and privacy protection, and runtime/compiler based deployment support.

- **Heterogeneous Distributed Data Management**: Distributed data management is key to facilitating analysis, machine learning, and data driven actuation and control in IoT deployments. To do so however, data replication, access and privacy control, and consistency protocols must account for varying resource scale, capacity, and lossy networks. Enabling such services in IoT settings requires coordinated research progress in hardware, protocols, and systems.

To enable this we have defined the following design principles, which we refer to as *SEALD*. Specifically, we believe that any end-to-end system capable of addressing this myriad of IoT challenges must be

- **S**tateless: computations will maintain no internal program state.
- **E**vent-driven: The fundamental programming abstractions will represent asynchronous events as first-class programming objects.
- **A**ppend-only: All storage abstractions will implement append-only update exclusively.

- **L**ogged: All events in the system will be logged for some amount of time after they have occurred and causal dependencies between events are tracked, facilitating replay.
- **D**istributed: The system will include abstractions for implementing distributed deployment, locality, and cross-domain trust.

In this document, we present an initial set of formal semantics for *SEALD*, to expose potential research opportunities.

# 1 Semantic Domains

$$p \in \mathit{Program} \qquad v \in \mathit{Value} \qquad c, d \in \mathit{Channel} \qquad i, j \in \mathit{Index} : \mathbb{N}$$

$$m \in \mathit{Media} : \mathit{Channel} \to \mathit{Value}^\star \qquad h \in \mathit{Handlers} : \mathit{Channel} \to \mathit{Program}$$

$$\mathit{log} \in \mathit{Logs} : \mathit{Channel} \times \mathit{Index} \to \mathit{Event}^\star \qquad e \in \mathit{Event} : \{\mathbf{get}, \mathbf{put}\} \times \mathit{Channel} \times \mathit{Index}$$

We use the Kleene star to denote sequences, e.g., $\mathit{Value}^\star$ denotes a finite ordered sequence of values. Descriptions of the semantic domains:

- *Program*: Programs that define individual processes. The syntax and semantics of these programs are left unspecified. The language is purely functional and contains the following terms: $c!v$ (send the value $v$ on channel $c$); and $c?i$ (retrieve the value of the $i$th element from channel $c$). There is an unspecified local semantics $\xrightarrow{\ell} \in \mathit{Program} \times \mathit{Program}$ that evaluates programs.
- *Value*: The values being communicated between processes.
- *Channel*: Names of channels between processes; the channels are modeled as sequences of values.
- *Index*: Indices of elements inside a channel.
- *Media*: Contains the state of all channels. Each channel name maps to a sequence of values that have been sent on that channel; these sequences are append-only with random-access reads. Given a value $v$ and a sequence $\vec{v}$, the notation $v \cdot \vec{v}$ denotes a new sequence equal to $\vec{v}$ with $v$ appended; the notation $\vec{v}(i)$ denotes the $i$th value in $\vec{v}$ or a default value if $\vec{v}$ does not have $i$ elements; and the notation $|\vec{v}|$ denotes the length of the sequence.
- *Handlers*: The event handler that gets invoked when a value is appended to a channel.
- *Logs*: Contains the event logs that record channel send and receive events, indexed by pairs $(c, i)$ that uniquely identify processes based on the channel and the index within that channel that caused that process to start executing.
- *Event*: A send event $(\mathbf{put}, c, i)$ recording that a value was sent on channel $c$ and became element $i$ within the channel; or a receive event $(\mathbf{get}, c, i)$ recording that a value was read from channel $c$ at element $i$.

# 2 Semantics

We define a global semantics $\xrightarrow{g}$ that abstracts over the local semantics $\xrightarrow{\ell}$ of program execution and mediates communication between programs. The signature of the global semantics $\xrightarrow{g}$ is:

$$\mathbb{C} \in \mathit{Configuration} : \mathcal{P}(\mathit{Program}_{\mathit{Channel} \times \mathit{Index}}) \times \mathit{Media} \times \mathit{Handlers} \times \mathit{Logs}$$
$$\xrightarrow{g} : \mathbb{C} \times \mathbb{C}$$

In other words, $\xrightarrow{g}$ nondeterministically maps between configurations consisting of a set of programs being executed, the state of the channels, the handlers for the channels, and the state of the event logs. Note that each program currently being executed is also labeled by the *Channel* × *Index* pair that initiated the program's execution (i.e., the program is a handler for the given channel and a value sent at the given index caused the handler to execute). This label uniquely identifies each program execution, even if the same program is being executed multiple times.

$$\frac{p \xrightarrow{\ell} p'}{(\{...p_{dj}...\}, m, h, log) \xrightarrow{g} (\{...p'_{dj}...\}, m, h, log)} \text{ LOCAL}$$

The LOCAL rule states that the programs are evaluated asynchronously using the local semantics unless they execute a $c!v$ or $c?i$ expression. Local evaluation has no effect on the channels or logs.

$$\frac{i = |m(c)| \qquad m' = m[c \mapsto v \cdot m(c)] \qquad \vec{e} = log(\langle d, j \rangle) \qquad log' = log[\langle d, j \rangle \mapsto (\mathbf{put}, c, i) \cdot \vec{e}]}{(\{...c!v_{dj}...\}, m, h, log) \xrightarrow{g} (\{...i_{dj}, h(c)_{ci}...\}, m', h, log')} \text{ SEND}$$

The SEND rule states that if a program evaluates a $c!v$ expression then the value $v$ is appended to channel $c$, the index at which it was appended is returned as the value of the expression, the log entry for the program's *Channel* × *Index* pair (uniquely identifying that particular program execution) is updated with a **put** event, and finally the set of programs being executed is expanded to include a new invocation of the handler for channel $c$ (labeled with the appropriate *Channel* × *Index* pair).

$$\frac{\vec{v} = m(c) \qquad v = \vec{v}(i) \qquad \vec{e} = log(\langle d, j \rangle) \qquad log' = log[\langle d, j \rangle \to (\mathbf{get}, c, i) \cdot \vec{e}]}{(\{...c?i_{dj}...\}, m, h, log) \xrightarrow{g} (\{...v_{dj}...\}, m, h, log')} \text{ RECEIVE}$$

The RECEIVE rule states that if a program evaluates a $c?i$ expression then the value on channel $c$ at index $i$ is returned as the value of the expression and the log entry for the program's *Channel* × *Index* pair is updated with a **get** event.

# 3 Enforced Properties

The model enforces the following properties for a SEALD system:

- Programs are purely functional in that the only persistent storage accessible by each program are the channels (which are append-only). Because the channels are append-only there are no memory synchronization issues between programs and channels; a *Channel* × *Index* pair uniquely identifies a persistent value (and also the program execution that was initiated by a send operation placing at value at that index).
- Channels are authenticated: each program knows exactly which channel it is interacting with.
- The event logs record a total ordering of send and receive events for each program execution and allow a global partial ordering of send and receive events across all channels that respects causal dependencies. In other words, the individual logs can be collated into a single partial order that can then be linearized into many total orders, but every such total order respects the causal chains of sends and receives.
- Information flow tracking. All information flow can be audited using the event logs as described above.

# 4 Checkable Properties

The following properties are not inherent in the model, but can be checked for on an application-specific basis (where an application is a set of channels and handlers):

- Usually programs should have bounded execution times. This can be checked statically by analyzing individual handlers or enforced dynamically using a timeout.
- An application may wish to enforce that there are no cyclic communication paths in the system. This can be statically checked by building a graph $G = (\mathit{Channel}, E)$ where an edge $c \to d \in E$ exists if any handler can read from channel $c$ and subsequently write to channel $d$ in the same execution. We can then check $G$ for cycles.
- An application may wish to enforce access controls by hiding some channels and handlers s.t. they can only be accessed by authorized channels and handlers. Any non-authorized handler can then only access the hidden channels and handlers via the authorized channels and handlers. This property can be statically checked by building the graph $G$ as described above and checking that the hidden channels can only be reached from the authorized channels.