RedSky: innovations for next-generation sensor networks

Zhiyuan Teo, Ken Birman, Robbert van Renesse
Department of Computer Science
Cornell University
Email: {zteo, ken, rvr} @cs.cornell.edu

1 Abstract

This white paper describes our RedSky solution, an emergent cloud-based, high-assurance platform for advanced sensor/actuator networks. RedSky is a comprehensive suite of software components that work together to deliver real-time monitoring and control of distributed sensor and actuator devices.

2 Introduction

As more and more everyday processes become automated and interlinked with one another, there is an increasing need for a distributed infrastructure that can address the control needs of a modern system with a large number of networked components.

What is desired is a high-assurance, scalable, secure system that can efficiently perform readouts of sensors, apply globally optimal actions and optionally archive the data for future use or offline analysis. Such a system should be robust to individual failure, self-healing and self-managing with respect to its aggregate resources. In addition, and quite critically, it should be easy to use and deploy. These are the end-goals for the design of our RedSky system.

Most applications requiring some form of distributed monitoring and/or feedback control can be improved through the use of RedSky. Some relevant examples include the electric grid, chemical processing, traffic management, patient monitoring and home automation, although clever and niche use cases could include software-only components such as file servers and distributed digital rights management.

3 System architecture

Our RedSky solution is hierarchical with three distinct levels: a sensor abstraction layer, a cloud data service layer and an application support layer.

Where possible, RedSky takes advantage of OpenFlow-enabled switches to quickly control and redirect data flows from point to point. However, the use of OpenFlow is not a critical requirement: in non-OpenFlow compliant networks, RedSky is still operable in software emulation mode, albeit with a slight performance penalty and the loss of a few features.

3.1 UUIDs as a naming system

Universally unique identifiers (UUIDs) are 128 bit numbers that are used to represent entities in the RedSky system. The goal of using UUIDs as the default mode of addressing within RedSky is to permit a scalable naming scheme that is compatible with existing IPv4 networks, while overcoming common issues pertaining to NAT traversal, DNS vulnerabilities and source authentication. In other words, practically no changes are required in an existing network infrastructure that is looking to upgrade to RedSky; RedSky will run harmoniously alongside other legacy network applications. However, in cases where some network infrastructure may be modified or setup from a scratch, the strategic deployment of specific RedSky technologies can improve performance and thwart DDoS attacks as well.

3.2 Sensor abstraction layer

Fundamental to the operation of RedSky are the sensors and actuators at the lowest levels of the system. These physical devices function as workhorses of the combined system, and are externally the most visible aspect of the system. Sensors collect data on behalf of the dependent upper layers in RedSky, while actuators perform actions as commanded from above. Optionally, each of these sensor/actuator devices may perform some low-level data-processing and act semi-autonomously.

3.2.1 Device properties

Every device in a RedSky deployment has several attributes: it has a UUID, an owner UUID, and an ordered list of UUIDs authorized to access the device, which we call the group access list. The owner of a device is typically the manufacturer of the device, or otherwise the UUID of the first point of contact when the device initially plugs into the RedSky network. This owner has the authority to manage the device remotely, which principally involves configuring and managing the group access list. An owner may not delegate or transfer its powers to another entity so owners have to be permanent identities on the RedSky network. However, local access to the device may be used to reprogram the owner UUID and/or group access list. In effect, this allows a local override if deemed necessary.

3.2.2 Devices and device drivers

Devices plug into the RedSky framework through user-mode device drivers. Block and stream devices are both supported. Device drivers run as standalone processes that bridge the gap between devices and the RedSky cloud computing layer. In effect, device drivers provide a standardized interface for data flow between devices and the cloud. Seen in another way, the device driver model in RedSky is analogous to that in actual operating systems, with an upper half managed by RedSky to interface with the 'operating system' (i.e. the cloud), and a lower half supplied by the device manufacturer that talks directly to the device.

Device drivers are written by linking device-specific I/O code with the RedSky device driver library, which is a powerful, flexible, cross-platform compatible C++ library that is straightforward to work with. For a device to work with RedSky, the programmer simply sets a few software switches to define the operation mode of the device and then registers a basic set of C++ function pointers that can be called upon to perform or respond to device-specific I/O events. All other aspects of the device driver may be controlled by the programmer; the RedSky device driver runs in a separate, non-intrusive thread that transparently performs the required duties – whether it is polling for a reading at every predefined interval, activating an upcall in response to a change in device status, programmatically shutting off a device to conserve power, or some other arbitrary action. In particular, RedSky does not hijack the `main()` function of the device driver, nor does it require the device driver to permanently hand off control at the end of initialization.

The RedSky device driver library handles all aspects of concurrency and communications with the cloud computing layer and presents a simple-to-understand view of accessing the device. All device accesses are atomic and serialized, meaning that no two registered functions may be run by RedSky concurrently; one function must complete in its entirety before another one may be started. This simplifies device management since no explicit locking mechanism is necessary.

3.2.3 Low-level controllers

For semi-autonomous device operation, some vendors may choose to run controller applications directly on the device itself. For example, a power management unit (PMU) in an electric grid may require very rapid control of the underlying power system using algorithms that do not depend on exterior feedback. In such scenarios, it is infeasible to deploy the control algorithms in an application far removed from the device itself. The RedSky device driver model allows for the deployment of such low-level controllers in two ways: managed and direct.

The managed method requires low-level controllers to register themselves with the RedSky device driver, optionally along with routines that define controller behavior (eg. startup, shutdown, query/modify state, etc). While not

critical to controller functionality, these routines permit higher-level manipulation of the devices, in effect allowing controllers and applications farther up the control hierarchy to enslave or override local applications such that decisions can be made for the greater good. As an added advantage, such controllers may effectively leverage the standardized RedSky device I/O model to arbitrate device access without implementing internal concurrency mechanisms.

Direct-deployed low-level controllers run outside of RedSky device management and are not subject to any RedSky limitation. In this method, controllers simply run their logic in the main thread of the device driver, relegating RedSky management to a secondary thread. While this is potentially more powerful and faster than managed mode, the programmer has to be careful not to let his controller code interfere with RedSky logic. Additionally, interprocess communications over the network may be limited without RedSky support; this is both an architectural and security feature.

3.2.4 Virtual devices and device hierarchy composition

Not all device drivers need to be paired with a physical device; they may also be multiplexed to one or more virtual devices, which themselves are device drivers. The RedSky device driver model allows each driver to manage its own devices, but also exports an interface that appears to RedSky as a device on its own. This permits other virtual device drivers to be written that manage multiple devices, and thus the construction of a device hierarchy.

The design of a device hierarchy is application-dependent, however in many use cases where a large number of sensors and actuators are present, it is infeasible for every device to forward data upstream for central processing. In these scenarios, it may be desirable to construct multiple levels of device management using virtual device drivers, where each successive level aggregates and summarizes the data of the preceding level before sending it higher up. Aggregation and summarization can be implemented in the low-level application logic within the virtual device drivers.

A device hierarchy is essentially a tradeoff between responsiveness and latitudinal visibility of devices. Device drivers (and their associated controllers) lower in the hierarchy are closer to the devices and can thus dispatch instructions or read devices with lower latency. However, they also manage fewer physical devices and can only make local optimizations within the subtree of its managed devices. Correspondingly, virtual device drivers higher up in the hierarchy are better able to make globally optimal decisions, but are less agile and have long instruction percolation times.

On the other hand, although data aggregation and summarization are potentially useful methods of reducing data explosion and network bottlenecks, it is not necessary to give up control granularity even in a hierarchical design. Device drivers may choose to enable data and instruction pass-through for certain physical devices, which enables control functionality to be implemented arbitrarily high up in the hierarchy. In corporate parlance, this allows micro-management of devices where macro-management is ordinarily expected.

3.2.5 Control conflict resolution

In device networks that do not adhere to a strict hierarchy (eg. mesh networks, ad-hoc networks, etc), readouts of sensors are relatively trivial however control conflict becomes a potential problem. A control conflict is the result of multiple control agents that simultaneously try to operate a shared device in non-compatible ways. It could be the result of a race condition among said control agents, or possibly a consequence of control elements making optimizations at different granularities.

In the case of conflicts, RedSky uses the device's group access list to decide control binding. The group access list establishes an order in which various entities can access and control a device. UUIDs farther up the list have preferential access and their actions may override those of UUIDs lower down in the list. If left unspecified in the group access list, the local UUID always has the highest preference. However, the owner UUID has to be explicitly named within the list for it to be considered within the control binding hierarchy. The latter two exceptions are to permit local overrides while encouraging the separation of responsibilities.

3.2.6 Disconnected operation

RedSky devices that lose all connectivity to the cloud are not precluded from normal operation as long as one or more control agents are still active within the disconnected network. In these situations, the devices continue to operate as per instructions from these control agents, but cloudbound data may be buffered until the cloud connection is restored. Buffering may be specified up to a certain limit within the device driver.

3.3 Cloud data service layer

One primary purpose of the cloud data service layer is to reliably and efficiently store and distribute data from the dependent devices. This cloud layer can be as small as a single node, or it can be a supermassive compute cluster with millions of nodes. Cloud nodes may be dedicated as with private ownership, or they may be shared in the case of community clouds.

Apart from buffering, storing and distributing data, the cloud data service layer is also responsible for resolving UUIDs to network addresses and routes.

3.3.1 Cloud structure

The cloud data service layer comprises two main subsystems: an externally visible front-end for the reception or dissemination of incoming data and a hidden back-end for replicated storage.

Front-end processes represent the entry point into the cloud, and consequently require some form of visibility from all nodes. This could be a public IPv4 address or a named node that is accessible within the RedSky routing infrastructure. An efficient cloud should typically have multiple front-end systems so that traffic loads may be shared and additional fail-over safety is provisioned.

Back-end processes are in charge of data replication (possibly to very distant sites) and storage. They provide the system with crucial data resilience properties, as well as offer a means by which available data can be retrieved for processing and state estimation.

While a conceptual difference exists between front and back-end systems, in actuality these are processes that can run on the same machine. An interprocess communication mechanism allows the front and back-end processes to be separated by arbitrarily large distances.

Front-to-back-end systems are typically matched based on network response times and are automatically managed by RedSky. The matching does not necessarily remain static however, and may be reconfigured dynamically according to changing system conditions. The exact mechanism of this reconfiguration is kept hidden from users and is completely transparent to both the sensor abstraction layer as well as the application support layer.

3.3.2 Naming and name resolution

The resolution of UUIDs to a destination is done by establishing a connection to a name server, which is a well-known IPv4 address. The exact server to connect to is selected from a pool of addresses initially statically defined at the requesting entity. The requester authenticates with the name server, and then proceeds to obtain routing information about the endpoint that it wishes to query.

The pool of IPv4 name server addresses stored with each entity can be updated dynamically. The name server itself may provide a new list to the requester, or in the case of a device, the device owner may propagate a new set of name server addresses.

3.3.3 Identity in community clouds

In allied cloud systems where several self-sufficient RedSky service providers decide to pool their resources together, a system for sharing identity and access control is required for cooperative function.

3.4 Application support layer

The application support layer is highest among the RedSky hierarchy and is a essentially a framework for centralized control and data processing. This layer abstracts away many of the messy details that pertain to cloud and device management, and provides a clean interface for users to read data, perform computation and request action.

Similar in spirit to the development of low-level controllers for devices, an application built to run atop the RedSky hierarchy needs to link with a RedSky cloud interface library. This library provides the necessary abstraction to communicate application intentions directly to the cloud systems and devices. Unlike low-level controllers, however, each application also has its own UUID. This UUID may be used in authentication as well as endpoint-to-endpoint communications with other applications and/or devices.

To facilitate careful computations that require temporal data, all accesses to devices and interprocess messaging via RedSky always include latency estimates as part of the return value. Latency estimates are broken into two components: software latency and network latency. Software latency measures the time between a RedSky component receiving a message and completing its processing. This takes into account queuing delays, command delays and hardware latency. For example, a slow device that has many read requests backed up on its command queue may exhibit very poor response times; this could be compounded by slow-moving queues on intermediate forwarding nodes. All of these delays contribute to the software latency measure. Network latency measures the approximate sum total time for messages in transit on hardware. It assumes symmetric links between nodes and the measure is derived from occasional ping/ack responses. Together, the latencies present an extra dimension to the data already gathered, and allows applications a means by which temporal synchronization can be attained. Applications can also use latency data to compute future command and transmission schedules.

3.4.1 Access control for devices

Applications are not unlimited in their scope to read or write devices in the RedSky hierarchy. Instead, their access is governed by discretionary access policies, which are expressed within the devices themselves.

Each device is initially preprogrammed with an empty access set. This access set can only be modified by the owner or through a local override on the device. The access set is a list of a pairings that specify UUIDs and their corresponding read/write access authorization. It is conceivable that the list may be extended to define the exact callback functions are authorized by an individual user.

3.4.2 Reading devices

Applications that have authenticated with RedSky and have read access to a device are allowed to perform state readouts on the device. These readouts may be obtained through one of two ways: an immediate read command execution or through the device cache. The application states its staleness tolerance for data readouts when it issues a read call; the device driver then checks its internal cache to find out if the most recent reading corresponding to an identical command is within that tolerance. If it is, the driver returns that reading along with the time elapsed since that reading was performed (ie. the staleness measure). Otherwise, or if the device does not support caching on this particular read command, the driver queues the read command for execution and returns the value when the appropriate read handler on the device driver completes.

3.4.3 Device domination and overriding low-level controllers

Authorized applications requiring read-access may read devices or their caches at any time provided they are not busy servicing a higher-priority entity, however writes to a device have to be mutually exclusive. Moreover, it is a frequent requirement that writes have to occur in correlated patterns, which necessitates the writing application gaining exclusive access to the device for a prolonged period of time. For this reason, applications may issue a device domination request, which functions like a pre-emptible lock. As long as an application holds the lock, no other entities with lower access priorities may write to the device. However, an entity with a higher access priority may dominate the device at any time and remove write access from the application. Device dominations are kept alive by

a ping/ack mechanism; the loss of this heartbeat signal over an extended time causes the lock to be automatically released.

Low-level controllers inherit priorities directly from their paired device. In the case of managed controllers, the application may choose to override these controllers by sending them requests to relinquish control. Control is only relinquished insofar as device domination is in effect; however an application can always issue requests to alter the state of the controller so that it continues to behaves differently after device domination loss.

4 Conclusion

We believe that RedSky successfully innovates the required technologies to implement a cohesive and complete solution for modern mission-critical sensor networks. Our platform offers the world's most advanced realtime distributed data processing capability and is unparalleled in comprehensiveness and flexibility. Together with its security and ease of use, RedSky is poised to define the next frontier in computing.