A Secure Architecture for the Range-Level Command and Control System of a National Cyber Range Testbed

Michael Rosenstein, Frank Corvese

Applied Visions Inc., Secure Decisions Division
{michael.rosenstein, frank.corvese}@securedecisions.com

Distribution Statement A – Approved for Public Release; Distribution Unlimited

ABSTRACT

In recent years, cyber security researchers have become burdened by the time and cost necessary to instantiate secure testbeds suitable for analyzing new threats or evaluating emerging technologies [1]. To alleviate this, DARPA initiated the National Cyber Range (NCR) program to develop the architecture and software tools needed for a secure, self-contained cyber testing facility. Among NCR's goals was the development of a range capable of rapid and automated reconfiguration of resources, broad scalability, and support for running simultaneous experiments at different security levels [2].

In this paper we present our architecture for the Range-level Command & Control System (RangeC2) developed as part of the Johns Hopkins University Applied Physics Laboratory's implementation of the NCR [3]. Our discussion includes the RangeC2's functional and non-functional requirements, the rationale behind its partitioning into layered subsystems, an analysis of each subsystem's fundamental mechanisms, and an in-depth look at their processing paradigms and data flows.

To meet the demands of this range, the RangeC2 was required to perform three primary jobs: 1) management of all range resources; 2) management of numerous concurrent experiments; and 3) enforcement of each experiment's resource security and perimeter isolation. Our discussion of the architecture will show how these requirements were met while overcoming the RangeC2's most critical challenges.

1 INTRODUCTION

Considerable amounts of human and financial resources are needed to assemble and configure cyber experiment testbeds. The reasons for this are numerous, and include testbeds designed to run only one experiment at a time, at a single security level, and with support for only manual configuration of hardware and software resources [1]. In response to these and other challenges, DARPA initiated the National Cyber Range (NCR) program to design and build the next generation cyber testing facility. From its inception, the NCR was designed with security and isolation as its highest priorities. Nowhere did this apply more in the Johns Hopkins University Applied Physics Laboratory's (APL) implementation of the NCR, also known as the Cyber Measurement and Analysis Center (CMAC), than in the Range-level Command & Control System (RangeC2), which was charged with the management of all range resources and each experiment's resource requirements. The RangeC2 also accomplished several other goals, including rapid and automated reconfiguration of the range's physical connectivity, support for simultaneous, physically-isolated experiments at different security levels, and oversight of processes that allowed freed resources to be sanitized and used again in other experiments.

To achieve these goals, the RangeC2 was required to manage a Layer 1 Switch (L1 Switch). The OnPath UCS 2908 L1 Switch [4] was selected for its ability to provide physically-isolated connectivity between any ports. In general, L1 Switches behave similarly to network patch panels, but include redundancy, diagnostic capabilities, and the ability to be dynamically configured through remote administration.

The RangeC2 was also required to support additional range activities including: 1) inventory management, with authority over what resources were on the range and if they were in service or offline; 2) administration and presentation of resource states such as available, assigned, or resetting; 3) experiment import, which stored lists of required resource types and scheduled their availability; 4) physical management of experiment resources; and 5) administration and presentation of experiment states such as building, running, or destroyed.

Because CMAC simultaneously managed resources and experiments at different security levels, the RangeC2 was required to always run at the system-high security level. For CMAC to run each experiment at its own security level, the responsibility for managing and executing those experiments was divided into two parts: the RangeC2, which managed resource assignment and physical connectivity across all experiments, and an Experiment Command and Control System (ExpC2), which ran at the same security level as the experiment it was part of, while managing the remaining aspects of the experiment's execution. To facilitate this, the RangeC2 logically grouped all resources used by an experiment into a "container." Then, as part of the RangeC2's management of an experiment, it could: 1) build a container by assigning available resources to it, including an ExpC2; 2) configure a container by altering the connectivity among resources within it, or removing individual resources from it; and 3) destroy a container by unassigning all resources from it. We refer to these groups of resources as containers because the RangeC2, for reasons of experiment isolation, had virtually no visibility into what was happening on those resources, making the fact that they were hosting cyber experiments irrelevant. It is therefore containers, not experiments, which defined the boundaries of isolation the RangeC2 was required to enforce.

For the remainder of this paper, we will refer to the RangeC2's management of containers rather than experiments, using the term "experiments" only when discussing items specific to an experiment's implementation.

2 RANGE C2 CONTEXT

The RangeC2 was required to communicate with several other CMAC components. The boundaries between these components were derived to maximize the isolation of each system while also providing increased levels of physical security to its most critical components. Figure 1 shows both the communication paths and physical security design of CMAC as it related to the RangeC2.

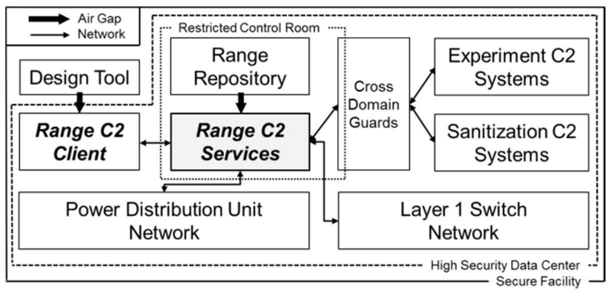

Figure 1 – Communication and Physical Security

The Design Tool, used both by range staff and external entities to generate experiments, would output several files detailing an experiment's configuration and execution. When complete, one of these files would be burned to a DVD and delivered to the RangeC2 Client, which imported the experiment into the RangeC2.

The Range Repository was a multi-level-security system that maintained a listing of the range's inventory, divided according to each resource's security classification. Whenever changes were made to the range's inventory, updates were made to the Range Repository's files and then burned to a DVD for delivery to the RangeC2.

The Power Distribution Unit (PDU) Network was an isolated network connecting the RangeC2 with the PDUs attached to each resource on the range. These PDUs provided the RangeC2 with control over each resource's powered state. The L1 Switch Network was an isolated network that connected the RangeC2 to the L1 Switch.

The Cross Domain Guards (CDGs) were specialized nodes that allowed only a basic set of predefined information to flow between the RangeC2 and the containers. The information included only what was needed to build, configure, and destroy a container, or sanitize a resource.

The ExpC2s were specialized resources that were included in containers to execute an experiment. They understood the functionality provided by the RangeC2 and could issue requests to it and react to commands from it via the Cross-Domain Guards.

Lastly, the Sanitization C2s (SanC2) were another type of specialized resource, possessing the same RangeC2 communication capabilities as the ExpC2s but designed for the sole purpose of resetting other resources so that they might be used again elsewhere.

Figure 2 demonstrates how these components worked together by providing an example of the range's state during normal operations. Here, the RangeC2 has assigned resources A, B, and ExpC2 to one container and resources D and SanC2 to another; resource C is unassigned [gray areas represent containers]. The RangeC2 configured the connectivity (dotted lines) among resources within a container, via the L1 Switch, but did not create connections that crossed a container's boundary. Additionally, the RangeC2 controlled power to each resource via the PDU network, and communicated with the C2s in each container via the CDGs (dashed lines).

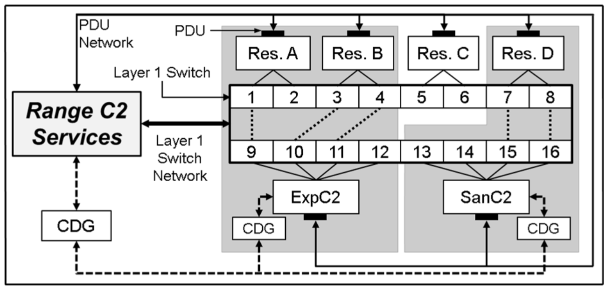

Figure 2 – The range's state during normal operations.

3 ARCHITECTURAL OVERVIEW

The RangeC2 was divided into four subsystems spread across three hosts. The subsystems were the RangeC2 Client, Range Controller (RC), Transaction System (TS), and Action Processor (AP); the latter two known collectively as the Switch Controller (SC). The rationale behind this distribution was security-based, and centered on four primary goals: 1) prevention of cascading failures, as each subsystem was designed to shield itself from errors occurring elsewhere; 2) division of responsibilities such that no single subsystem could plan, initiate, and execute a task (defined as a logical function of the RangeC2) without the cooperation of others;

3) layering of physical security such that the subsystems with the most control over the range's state were on the most restricted hosts; and 4) separation of the RangeC2's most critical activities (e.g. state management, container isolation, and range configuration) from its other less rigorous jobs as a way to reduce complexity and isolate the Certification and Accreditation (C&A) efforts. Figure 3 shows the subsystems that comprised the RangeC2, their interconnectivity, and connectivity to other components.

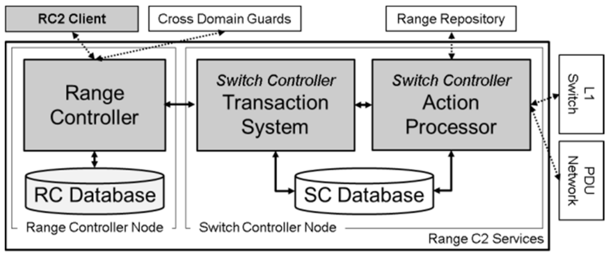

Figure 3 – Range C2 Partitioning

The RangeC2 Client was the user interface for the RangeC2. It allowed users to import experiments, view the range's schedule, request the building and destruction of containers, view current container and resource states, request resources be moved online/offline, and review system logs and alerts. Because it needed to be accessible, the RangeC2 Client was situated on its own host within the range's high-security data center, allowing it to be accessed by all range personnel. The client was designed to run intermittently and contain no persistent data, forcing it to rely on the RC for the data it displayed and for the handling of its user's requests.

The Range Controller was the RangeC2's system-level processing engine and communications hub. It was responsible for scheduling containers, performing logical-to-physical mappings (mapping abstract resource types needed by an experiment to available resources that could be assigned to a container), and historical record keeping of resource usage and maintenance. Because it was connected to the RangeC2 Client, the ExpC2s and SanC2s (via the CDGs), and the SC, the RC was also responsible for handling and responding to requests made by these systems. The RC could not fully process requests that affected the range's state, however, making it a buffer between the SC (which could) and the other range components, which were assumed to always be untrusted. To strengthen its buffering capability, the RC was situated on its own host within the range's restricted control room, where access was limited to only a small subset of range personnel. To support the remainder of its activities, the RC ran continuously, and accessed a private database.

The SC was responsible for knowing the exact state of the range at all times (e.g. resource interconnectivity, assignment, and power; container state; resource state; etc.) and securely affecting changes to that state when requested and allowable [as the only part of the RangeC2 capable of affecting change, the RangeC2's portion of the C&A effort was targeted towards this subsystem]. Specifically, the TS would receive task-specific requests from the RC, determine how they should occur (including all necessary security, assignment, and isolation checks), and then initiate their processing. The AP then processed each task, recording the results and updating the range's state at every step. Additional functions, such as auto-initiated consistency checks and status request fulfillment, were also handled by the SC. Because of their tight coupling, the TS and AP were both situated on the same host within the range's high-security data center and shared a common, private database. These subsystems also ran continuously.

4 RANGE CONTROLLER (RC)

To manage the flow of range operations, the Range Controller was charged with making decisions that affected how the range behaved at the system level. This included handling its normal range responsibilities and reacting to various process failures, such as when the SC was unable to complete a requested task due to security constraints or the failure of an external system. Although "security aware", the RC's primary goal was to support the usability needs of the range.

The RC was designed as an asynchronous, event-driven system. It received stimuli from external entities (in the form of RangeC2 Client, ExpC2, and SanC2 requests, or SC responses) and then processed them within event handlers, one per stimulus type. After processing a stimulus, each handler culminated with either an intermediate request to the SC or a response sent back to the requestor. Each handler was also thread-safe so multiple stimuli could be processed simultaneously, even if they were of the same type. To achieve this, the RC created worker threads for each stimulus that arrived and then executed the appropriate handler on that thread. Once the handler's processing had completed, the thread was terminated and all context associated with the event discarded. As a result, when a handler culminated with an SC request, it stored enough information in the RC's database to allow the response's handler to understand the original request conditions. The advantage to this mechanism was that if the RC were taken offline while the SC was processing its request (a sometimes lengthy activity), the RC could still handle the SC's response once it came back online.

Figure 4 provides an overview of the RC's processing flow and is used to support the following example, which demonstrates how the RC functioned. In this example, an ExpC2 requests that the RangeC2 alter the physical topology of its container.

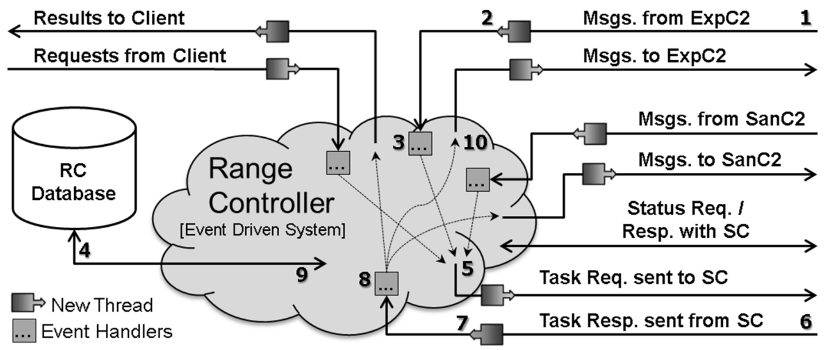

Figure 4 – Range Controller Processing Flow

Initially, the ExpC2 in container `1701` sends a message, or stimulus, to the RangeC2 (via the CDGs, which verify each message's origin) requesting that its container's topology be changed (1). The ExpC2 also sends a request ID unique to itself (EC2RID) in the message. The RC receives the request, and fires a `TopologyChange` event on a new thread (2) with the associated information. Next, the RC's "Topology Change Event Handler" receives the event and begins to process it (3). To complete the request, the handler determines it must send a `LOAD_PHYSICAL_TOPOLOGY` request to the SC because the RC cannot directly alter the L1 Switch on its own. To do this, the handler first generates an Internal Request ID (IRID) [`d9e6-RC2SC`], then stores information about this event in the RC's database (4), and then finally sends the request to the SC for processing (5). Once sent, the RC's part in this request's processing is complete, so the handler's thread is terminated and its in-memory state released.

Later on the RC receives a response from the SC (6) with an IRID of `d9e6-RC2SC` stating the `success` {or `failure`} of the request. Based on this, the RC fires a `RequestComplete` event on a new thread (7), again with the associated information. Next, the RC's "Request Complete Event Handler" receives the event (8) and uses the IRID to recall the information previously stored about this request (9). The handler then uses this information to finish processing the original request, which in this case only requires the handler to send a response containing the original EC2RID and the result of the request back to the ExpC2 in container `1701` (10), via the CDGs.

5 SWITCH CONTROLLER (SC)

The Switch Controller made changes to the range's state, both logically (i.e. the RangeC2's internal understanding of the system's configuration) and physically (i.e. the range's hardware configuration), in a way that strictly enforced security level constraints and the isolation of containers. In fact, the SC had no interest in the success or failure of those tasks requested of it; rather it considered security, isolation, and state management its paramount goals. Any requests that violated these principles were halted, marked as failed, and left to the RC to determine how they affected the range's overall operation.

To achieve these goals the SC needed to keep a persistent record of the range's exact state, even when it was being changed, as that record was the basis for all decisions made by the system. To track the range's state while changes were happening, the SC kept a record of its own processing state so that if it went offline unexpectedly, the SC could be brought back online in the same state it was in when it had failed. Without this, partially completed tasks would have led to disparities between the range's actual state and its recorded state, causing the SC to make decisions based on flawed information. This capability also provided continuity for higher-level activities, as all requests completed their processing and provided a result, even if system failures occurred along the way. The SC also provided status to the RC whenever necessary, and initiated routine system integrity checks to verify the accuracy of recorded state.

5.1 TRANSACTIONS AND ACTIONS

At the core of the SC was a two-phased approach to processing requests. The first phase, managed by the Transaction System, involved predetermining how requests should be executed; the resulting list of steps was known as a *Transaction*. The second phase, managed by the Action Processor, involved the monitored and recorded execution of each step, or *Action*, contained within a transaction.

5.1.1 TRANSACTIONS

Transactions represented the SC's methodology for performing individual tasks. During the SC's development, transaction templates were created for each task the SC supported. These templates provided the rules for generating the list of actions needed to accomplish a task, but required specific information, such as an ID or current range state, to actually produce that list. When a specific task was requested, the corresponding template was invoked by passing in the required information, which then built an instance of the desired transaction. Fundamentally, these instances were nothing more than sets of instructions (actions), and were themselves decoupled from the mechanisms responsible for carrying them out. Figure 5 demonstrates this process.

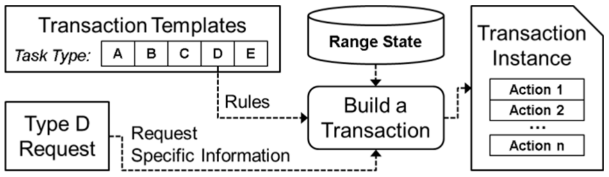

Figure 5 – Transaction Build Process

Many of the transaction types supported by the SC corresponded directly to requests received from the RC. Referred to as request transactions, some examples of these transaction types included: `INVENTORY_UPDATE`, `INSTANTIATE_CONTAINER`, `LOAD_PHYSICAL_TOPOLOGY`, `TEAR_DOWN_CONTAINER`, and `MOVE_RESOURCE_OFFLINE`.

After a transaction was built, both its metadata and associated actions were stored in the SC's database; once stored, the transaction was submitted for processing. One of the metadata attributes vital to tracking the progression of a transaction's execution was its processing state. Table 1 describes these states.

Table 1 – Transaction Processing States

State	Description
QUEUED	Processing has not yet started. This was the initial state of the transaction when stored in the SC's DB.
PROCESSING	Processing has begun, but not yet finished.
FAILED	Indicates that an action resulted in an undesirable state. This includes process failures, such as a failed check or external system failure, and processing failures, such as an error in the code.
SUCCEEDED	Indicates that all actions completed successfully.
FAILED-ACK	Indicates that a FAILED transaction has been handled by the Switch Controller.
SUCCEEDED-ACK	Indicates that a SUCCESSFUL transaction has been handled by the Switch Controller.

One might conclude from Table 1 that transactions either succeeded or failed, but this is not entirely correct. Transactions had to be built before they were executed, and that process could have failed as well. Circumstances that led to this type of failure included invalid information passed into a template (e.g. non-existent IDs), code errors, or the failure of any preliminary checks, which were assertions in each template that helped predetermine if a transaction was destined to fail during execution. Early identification of these transactions, prior to transaction execution, was preferred because the recovery process at that stage was much simpler than for executing transactions. It's worth noting that while some preliminary checks were conclusive, many were not, because the state of the range was subject to change between the time a transaction was built and the time it was executed. Accordingly, preliminary checks only prevented a transaction from executing, which meant that each condition tested as part of a preliminary check had to be re-tested at the time of execution.

Each request made of the SC could have one of three possible results: failure to build, failure to execute, and success. The SC was required to handle all three outcomes for each transaction type. This led to the development of a transaction flow matrix (TFM), which described the specific activities the SC should take in response to each result. For transactions that were successfully built, achieving either the FAILED_ACK or SUCCEEDED_ACK state required that all responses dictated by the TFM be completed first. Table 2 lists the response types available to the SC.

Table 2 – Transaction Flow Matrix Responses

Response	Description
Success to RC	Report the success of this transaction to the RC.
Failure to RC	Report the failure of this transaction to the RC.
Do Nothing	Take no further steps.
Raise Alarm	Raise an alarm, which was a tool for reporting errors outside the linear flow of processing.
New Transaction	Start a new transaction in response to this one.
Initiate Reset Process	Initiate the reset process, which was responsible for sanitizing resources.

Often, the failure of an executing transaction led to the invocation of another whose sole purpose was to clean up after the failed one. These recovery transactions brought the range back to normal operation by identifying where the failure occurred and taking the necessary steps to reverse the process; sometimes this was simple, but other times required resources to be disconnected, sanitized, or forced offline. Since recovery transactions altered the range's state, they too were subject to the full transaction lifecycle, including being part of the TFM.

5.1.2 ACTIONS

Actions were the atomic units of work that the SC could perform. By atomic we mean that each action existed independently of all other actions but was itself indivisible. By indivisible we mean that if an action's unit of work was attempted, regardless of success, the results had to be recorded. While transaction templates were aware of the action types they could enlist, actions were transaction agnostic, focusing only on their work and therefore accessible to every transaction in any context. Some examples of action types included `New-Container-Assignment`, `Power-On-Resource`, `Break-L1-Connection`, and `Set-Resource-Dirty`. Although some actions made physical changes to the range, most either altered its logical state or asserted various conditions. It was through these "assertion" based actions, in coordination with strict rules governing their execution, that the system was made secure.

Like transactions, each action also had metadata and a processing state associated with it. Table 3 describes the different processing states an action could be in, and how they were used to recover from situations in which an action was terminated part way through its execution.

Table 3 – Action Processing States

State	Description
QUEUED	Processing has not yet started.
QUEUED-INITIALIZED	Dispatched for execution, but not yet executing. Prevents other actions from being dispatched.
RECOVERY-INITIALIZED	Equivalent to QUEUED-INITIALIZED, but knowing that this action originated from the RECOVERY state.
PROCESSING	Action is being executed. This was the only state where an action's indivisibility was in question, so time in this state was minimized. If an action enters this state from the RECOVERY-INITIALIZED state, it knows that it was previously running and must now pick up where it left off.
RECOVERY	Was previously PROCESSING but interrupted mid-way due to a shutdown or system failure.
FAILED	The action's work encountered an error or failed an assertion, and the results were recorded.
SUCCEEDED	The action's work completed successfully and the results were recorded.

Figure 6 shows the transitions between an action's processing states. The dotted lines represent transitions that occurred whenever the system was brought online.

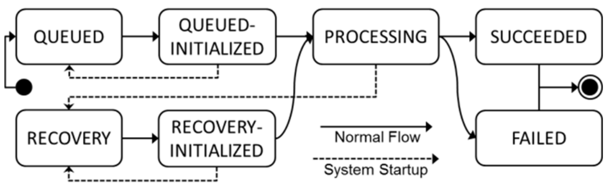

Figure 6 – Action Processing State Transitions

5.1.3 ADDITIONAL FEATURES

Progress, Logging, & Forensics: By storing the actions associated with each transaction prior to execution, and updating each action's state during execution, the SC was able to easily generate structured logs on-demand for each transaction on the range. Logs generated prior to a transaction's completion also detailed the transaction's progress by showing which actions were completed, processing, or still queued. Resource-based and container-based logs could also be generated, detailing every transaction, and component action thereof, which occurred during their respective lifetimes.

Independent Review: As a candidate for C&A, and as the system most responsible for security and isolation in the RangeC2, the ability of the SC's algorithms to be independently reviewed was of paramount concern. To achieve this, the transaction/action paradigm drew clear lines, both logically and in code, between the processing engine, the rules defining how tasks occurred, and the mechanisms used to perform the work. So, instead of a code base that interwove these concepts together, each template and action was self-contained and independent.

5.2 TRANSACTION SYSTEM (TS)

The Transaction System managed the flow of transactions through the SC. This meant accepting requests initiated internally or by the RC, building transactions based upon those requests, sending the transactions to the AP for execution, and processing the results through the TFM.

All communication between the TS and the RC took place within a limited API that existed only at the task and status request levels. By design, no mechanism existed for requesting more-granular activities from the TS, as that may have usurped the security restrictions it was responsible for maintaining. This feature, in combination with each transaction template's hard-coded rules and security checks, allowed the TS to handle any input, no matter how malicious or misguided, and still behave in a secure manner. This was a key consideration in focusing the RangeC2 portion of the C&A effort on the SC only.

At the center of the TS was a single "main" thread that managed the flow of all transactions. When requests arrived, they were placed with their attributes in a Task Queue (Task-Q), which stored them until they could be processed, and acted as a thread boundary making all requests asynchronous. Once in the queue, the main thread would retrieve the task, determine its type, and invoke the appropriate template. If building the transaction failed, the TFM would determine the appropriate course of action; if it succeeded, the transaction was stored in the SC's database and the AP asked to execute it. When the AP finished, it signaled the TS by placing a notification in the Response Queue (Resp.-Q). The main thread then retrieved this notification, loaded the transaction, and processed the results through the TFM.

Figure 7 provides an overview of the TS's processing flow and is also used to support the following example,

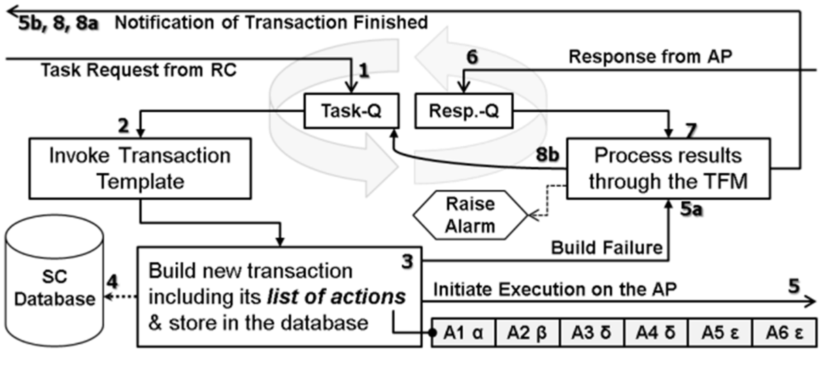
5b, 8, 8a Notification of Transaction Finished
Task Request from RC
1
6
Response from AP
2
Task-Q
Resp.-Q
7
Invoke Transaction Template
8b
Process results through the TFM
Raise Alarm
5a
Build Failure
SC Database
4
3
Build new transaction including its list of actions & store in the database
Initiate Execution on the AP
5
A1 α
A2 β
A3 δ
A4 δ
A5 ε
A6 ε

Figure 7 - Transaction System Processing Flow

which demonstrates how the TS functioned. This example continues where the previous one left off.

Initially, the TS receives a LOAD_PHYSICAL_TOPOLOGY request from the RC, with attributes detailing the request's desired topology, Container ID [1701], and IRID [d9e6-RC2SC], and places it in the Task-Q (1). Next, the main thread pops this request off the queue, determines its type, and passes the accompanying attributes into a LOAD_PHYSICAL_TOPOLOGY transaction template (2). Using the attributes and the range's state, the template determines the necessary actions required to achieve the desired topology change, which for this task includes (in order): Verify-Container-Instantiated (α)[confirms the container is running], Resource-Assignment-Check (β)[1 per resource; confirms the resource is assigned to the container], Break-L1-Connection (δ)[1 per current connection], and Make-L1-Connection (ε)[1 per new connection] (3). Once built, the transaction and its actions are stored in the SC database with all processing states set to QUEUED (4), the AP is notified of the new transaction (5) by its ID in the database, say 127, and the main thread moves on. Had the build process failed, nothing would have been stored in the database and the TFM would have handled the result (5a) by initiating a Failure to RC response (5b).

Later on the AP notifies the TS that it's done with a transaction by adding a response to the Resp.-Q (6). Next, the main thread pops this response off the queue, determines it was for transaction 127, and retrieves that transaction from the database. The TS then passes this information to the TFM (7) where its processing state, now set to SUCCEEDED, is used to determine how it will be handled; for this transaction type, a successful outcome only requires a Success to RC response (8). Once the notification (which includes the IRID) is sent, the transaction's processing state is set to SUCCEEDED_ACK and the TS's role in this process is complete. Had the transaction failed, the TFM would have performed both a Failure to RC response (8a) and a New Transaction response of type LOAD_PHYSICAL_TOPOLOGY_-RECOVERY using attributes pulled from the original transaction (8b).

While nearly all transactions occurred as a result of an external request or TFM response, maintenance transactions were initiated by the TS at regular intervals regardless of all other activities. Designed to be "self-checks", these transactions were tasked with detecting inconsistencies between the SC's record of the range's physical state, and the current configurations returned by the L1 Switch and PDUs at the time of the check. If a disparity was found, it resulted in the transaction's failure, which caused the TS to raise an alarm as this was an indication of potential range tampering. Maintenance transactions also queried these physical devices for their own alarms (e.g. a dying battery) and reported any issues found.

Finally, the TS also managed the reset process associated with each resource brought online or freed from a container. This activity was achieved through a series of sanitization transactions that brought resources back to a state where they could be re-assigned.

5.3 ACTION PROCESSOR (AP)

The Action Processor's singular purpose was to execute the actions contained within transactions according a set of rules, which governed the process:

1. All transactions that were part of the same container must be processed consecutively.
2. Transactions that were from different containers could be processed concurrently.
3. All actions within a transaction must be executed consecutively and in sequence order; if an action fails, processing of that transaction fails and any remaining actions go unexecuted.
4. The same action type may be executed concurrently, from separate transactions, unless otherwise specified by an action lock.

To adhere to these rules, the AP divided all transactions into two categories: those that existed as part of a container, such as LOAD_PHYSICAL_TOPOLOGY, and those that did not, such as MOVE_RESOURCE_ONLINE. Then, all transactions considered not part of a container were assumed to be in a single "null" container.

At the center of the AP was a main thread that supported the execution of transactions and actions. When notification of a new QUEUED transaction arrived, the main thread retrieved that transaction from the SC database, and if it was not from a container already having another transaction processed, it would be added to the AP's list of currently processing transactions and marked as PROCESSING in the database. Once set up, the transaction's first action was marked as QUEUED-INITIALIZED and dispatched for execution on a new worker thread. When the action was ready to perform its unit of work, the action was marked as PROCESSING, the work occurred, and then the action was marked as SUCCEEDED or FAILED, depending on the result. Upon completion of an action, notification was sent to the Action Queue (Action-Q), where the main thread retrieved the notification and processed the results according to three rules:

1. If the action succeeded and there were more actions in the transaction, the next action was dispatched.
2. If the action failed, the transaction was marked as FAILED and processing of that transaction was halted.
3. If the action succeeded and there were no more actions in the transaction, the transaction was marked as SUCCEEDED and its processing was halted.

Once a transaction's processing was halted, the main thread removed it from the list of currently processing transactions and notified the TS of its completion. Finally, the main thread queried the database for QUEUED transactions from the same container as the transaction that just completed; if one was found, it was set up – repeating the process.

Figure 8 provides an overview of the AP's processing flow, and is also used to support the following example that demonstrates how the AP functioned. This example continues where the previous one left off.

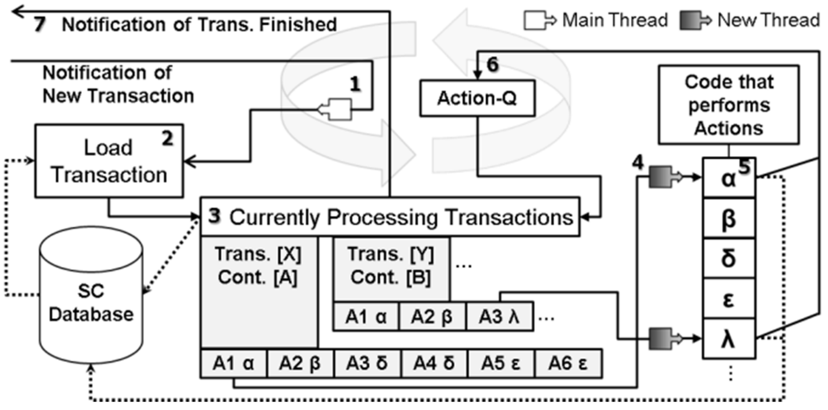

Figure 8 - Action Processor Processing Flow

Initially, the AP receives a notification of a new transaction with ID `127` (1). The transaction is retrieved from the database and determined to be part of container `1701` (2). The AP reviews its list of currently processing transactions and concludes that no other processing transactions are part of that container. As a result, transaction `127` is added to the list (3: `X=127, A=1701`) and its processing state is set to PROCESSING. Once added, transaction `127`'s first action, with a type of `Verify-Container-Instantiated` (α) is dispatched for processing by setting its processing state to QUEUED-INITIALIZED and then transferring its execution to a new worker thread (4). From there, the action enters the block of code responsible for executing actions of this type and it is executed (5). To do this, the action is first marked as PROCESSING, then the check takes place (Container `1701` is running), and finally the action is marked as SUCCEEDED. Once complete, a notification is sent to the Action-Q (6) where the main thread retrieves it, looks at the result of the action and the transaction it originated from, sees there are still more QUEUED actions in transaction `127`, and dispatches the next one. Once the last action completes, transaction `127` is marked as SUCCEEDED, removed from the list, and a notification is sent to the TS informing it transaction `127` has finished (7). Had another transaction from container `1701` arrived while this one was processing, it would now be given an opportunity to execute.

Although most actions could execute concurrently as part of separate transactions, some were subject to mutexes (action locks) that prevented more than one thread from accessing a set of actions at the same time; some action locks affected only a single action, while others affected multiple actions. For example: one action lock wrapped just the `New-Container-Assignment` action, which both checked the availability of resources and assigned them to a container. Had multiple threads been able to execute this action concurrently, race conditions may have led to the assignment of a single resource to multiple containers – a clear violation of container isolation. Other action locks wrapped multiple actions, such as the one that wrapped all actions that communicated with L1 Switch. This prevented 1) multiple requests from being sent to the L1 Switch at the same time, and 2) changes to the L1 Switch configuration during consistency checks.

6 ALARM SYSTEM

Integrated throughout all subsystems of the RangeC2 was an alarm system that allowed any code path to report an error, inconsistency, or unanticipated result through channels outside the linear flow of processing. Under the premise that no issue should ever be ignored, it recorded and propagated to the end user every issue (alarm) raised so that each alarm's potential impact on system integrity could be analyzed and appropriately handled. With every alarm came a time-stamp, origin, severity, technical description of the issue, and a human-readable analysis describing the context of the issue and how it should be addressed. In general, there were five circumstances that raised an alarm: 1) unanticipated exceptions [exceptions thrown in a context not anticipated by the developers]; 2) severe errors [errors accounted for, but indicative of larger problems]; 3) recovery failures [TFM's response to a FAILED recovery transaction: since initiated internal to the SC, this was the only mechanism for expressing their failure]; 4) system maintenance issues [similar to recovery failures, but indicative of serious problems or malicious behavior]; and 5) identification errors [messages sent to the RC from unrecognized sources].

7 LESSONS LEARNED

The system described was developed over a sixteen-month period between Jan. 2010 and Apr. 2011. Overall, it proved adaptable to change and required no deviations from the architecture in order to complete.

During its development, early adoption and rigorous adherence to external interfaces were pivotal to the completion and integration of this system. Use of the Agile methodology also proved beneficial by defining achievable goals, such as building a container or changing its topology, and allowing routine integration of the system. The most remarkable aspect of the system, however, was its ability to prevent logical errors by separating the logic from the execution, as described in section 5.1.3.

One element of the system's design we would like to have changed was the requirement that messages to/from the SanC2 be bridged through the RC. Although this provided the buffer described in Section 3, it also required that information related to sanitization pass through systems outside of the SC, SanC2, and CDGs. Additionally, the SC might have benefitted from a new transaction processing state representing the "building" phase. This state would have reduced the time between a requests arrival at the TS and its initial persistence.

8 REFERENCES

1. **Michael VanPutte, Ph.D., CISSP.** NCR - The Future of Cyber Testing & Experimentation. https://acc.dau.mil/adl/en-US/386254/file/52147/MVP NCR Test Week 100524 V4.pdf.
2. **DARPA.** National Cyber Range. *DARPA - STO.* www.darpa.mil/Our_Work/STO/Programs/National_Cyber_Range_(NCR).aspx.
3. **JHU/APL.** APL Receives $24.7 Million to Build Prototype Cyber Range. www.jhuapl.edu/newscenter/pressreleases/2010/100121.asp.
4. **OnPath Technologies.** UCS™ 2900 Connectivity System. www.onpathtech.com/products/2900-series/.

Disclaimer: The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.