GETS Engineering and Development Center

Introducing Canon Event Database

14 January 2014

Steven Siebert
GETS Team
Contractor, T-3 Solutions LLC
steven.siebert@t-3-solutions.com

Table of Contents

(U) Preface

(U) This paper is the first in a series of articles exploring the use of the ~~CubbyDB~~Canon event database to meet common requirements in the development of applications for the Federal Government and Intelligence Community (IC). These common requirements are, in part, derived from security mandates from the National Institute of Standards and Technology (NIST) and the Defense Information Systems Agency (DISA) application development guidelines, as well as environmental aspects in developing distributed applications for use within the Intelligence Community (IC). While these articles focus on the impact of developing applications in this environment, many applications outside of the IC, including commercial applications, share similar requirements.

(U) ~~CubbyDB~~Canon has been developed as an Open Source Software (OSS) by US Army G2 as part of the GEOINT Enterprise TPED Services (GETS) project. ~~CubbyDB~~Canon is made publicly available under the Apache V2 License and can be found at http://github.com/geoint/canon~~cubby~~.

(U) Abstract

(U) ~~CubbyDB~~Canon is an *event based database* designed to support rapid implementation of development enhancements to GETS. Its unique approach for satisfying nonfunctional system requirements allows developers to focus the business processes that enhance user interfaces and optimize system performance. ~~CubbyDB~~Canon provides a dynamic mechanism for making user requested or system required improvements to GETS without impacting current capabilities or operations. This approach has the potential to provide great efficiencies in the development of new applications for the US Government, the Department of Defense, and the Intelligence Community.

(U) Background

(U) It is common to build an application architecture based on events. From graphical user interfaces to service oriented architectures, events follow a familiar paradigm. Those models typically use events as transient (often in-memory only) artifacts that are discarded after the event observer has successfully processed the event. Much of the time, processing of an event includes the storage of data, where the changes identified in the event are merged with current state. In other words, the data contained within an event is used to update existing data. So, data is stored as a current "snapshot", only depicting the current state. For most commercial applications, this is adequate, but applications within the IC often have interesting requirements that make this model inadequate.

(U) Influences placed on IC system requirements are not only functional, but include security, policy, and unique network constraints that must be considered for every application. Before the first user story is recorded, applications must meet hundreds of requirements leveraged by external influences. Many of these requirements are met, or mitigated, by additional application code implemented within each application, or as integration projects, deployed across the community. This approach not only adds additional cost, complexity, and risk to initial project development, it also increases the Operations and Maintenance (O&M) overhead induced by system complexity and increased occurrences of bugs within applications.

(U) The two main categories of requirement when planning an application are functional and non-functional. Functional requirements mainly detail the business requirements the particular software needs to accomplish. Examples may include the ability to assign tasks to individuals, or the ability to review the results from a dirty-word scan. Non-functional requirements are those requirements that do not directly impact the user's interaction with the system. Non-functional requirements are often focused around system aspects such as user load requirements, response times, security, scalability,

code maintenance, delivery schedule, etc. The design of ~~CubbyDB~~Canon, and related architecture, focuses on satisfying the non-functional requirements common to IC applications. By doing so, developer time is spent focused on the functional capabilities of the application, rather than re-implementing the same non-functional requirements in every application.

(U) The ~~CubbyDB~~Canon Event Database is a result of careful analysis of these influences and their requirements. The resultant database, and the corresponding application architecture making use of ~~CubbyDB~~Canon, inherently satisfies many of these requirements, which are detailed below. By satisfying these common requirements through architecture and at the database level, application developers are able to focus on the business problem at hand. Applications designed with ~~CubbyDB~~Canon result in smaller decoupled components, which reduces the time to develop and accredit the system.

(U) Traditional Application Architecture

(U) Traditional application architectures centralize data storage in one main database where one or more applications can directly access the data. Most enterprise architectures have a single application that accesses its data and provides access to that data through a set of web services. Regardless of the size of the application, this architecture creates data silos, which often require custom web services for access (see Figure 1). Applications that integrate with these web services are limited to observing the data after the application applies its business logic (i.e. what if the original application had a bug, all downstream data systems now contain invalid data). They are also limited by the depth and variety of web services the application provides.

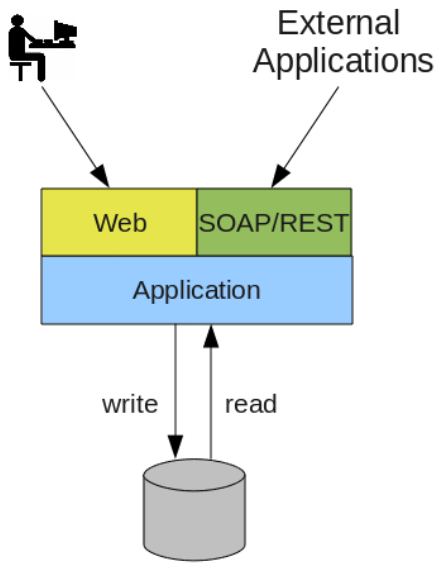

Figure 1: (U) Traditional Application Architecture

(U) Data Consistency

(U) There are several reasons that centralizing data in a single data store is popular, but the most important reason is to assure data consistency. Though different data stores provide different features to support scale-out or specialized search capabilities, they all must ensure data consistency to be a viable

product. Consistency is quite difficult, and sometimes impossible, if the data integrity requirements are spread across multiple systems that must be tied together at the application level.

(U) While the relational database remains the defacto data storage system for most applications, many applications have started using specialized "Not-Only SQL" (NoSQL) databases to mitigate the capacity and functional limitations of relational databases. In practice, the centralized data store model depicted in Figure 1 often remains the same, and specialized databases are frequently used as drop-in replacements for relational databases. While NoSQL databases have capabilities not available in a relational database, they bring a unique set of limitations that must be managed.

(U) Limitations of Data "Snapshots"

(U) Data "snapshot" representations, as they are stored in traditional databases, limit the way the data can be used. While this data can generally be considered 'true', it cannot tell you how this truth came to be. On the surface for most applications, this is not a problem. However, for IC systems there are - non-functional and functional requirements which demand that the lineage of the data be traced in a manner that is both useful for analysis and auditing. Applications can certainly be designed to use a relational database to meet these requirements, but that causes extreme inefficiency. Every application developer has to design, implement, and maintain the same functions in multiple ways.

(U) Data History: How did we get here?

(U) Data recorded as it is "now" is unable to tell the story of how it came to be in its current state. Without such history, all users of that data, from IA to analysts to mission planners, cannot answer questions they need to ask. For example, a mission planner may request metrics on the number of intelligence reports that were published against a specific requirement priority. Databases can be designed to answer this question, but the question itself is more complicated than it seems. To determine which intelligence publications were produced against a requirement priority, we need to look at what the requirement priority was at the time of report publication, not its current value. By managing data as it exists "now", the results of such a query will produce false results.

(U) New Application Features "Start Now"

(U) As applications evolve, new functional requirements for the application emerge. When features are added to applications, they can only access current 'true' data. Applications generally need to leverage processes that create data, and those processes are only available after deployment. Within the IC, not only do new capabilities *require current state*, they also need to know how information was added to the system. These types of components can start providing resultant data only after they are deployed. This creates enormous pressure during application development; when anything is changed due to a software bug, a requirement change, or an invalid test, all the process data begins anew.

(U) Application Architecture Using Event Storage

(U) Applications designed to use ~~CubbyDB~~Canon cannot follow the same traditional architecture and simply replace the centralized data store with ~~CubbyDB~~Canon. ~~CubbyDB~~Canon requires a shift in architecture design to provide the most benefit. Rather than a single data store providing both data consistency and all the read/query features necessary for an application, ~~CubbyDB~~Canon focuses on simply maintaining data as events. This means that data is stored as it enters the system, prior to business processing (aside from data validation). As a result, ~~CubbyDB~~Canon can answer questions regarding the history of data, how it was entered into the system, and also allow applications to

"replay" data so new application features can be "seasoned" by historic events. However, while ~~CubbyDB~~Canon it excels in data consistency and history, it is a terrible data store for applications that must return the current state of data. The ~~CubbyDB~~Canon architecture leverages event handlers to pass data to other data stores, which has significant implications in the way applications are developed and delivered.

(U) Unlike traditional application architectures, applications using ~~CubbyDB~~Canon separate writes and reads. All writes are written to ~~CubbyDB~~Canon; while all queries (reads) are conducted against any database or data store that best fits its needs (see Figure 2). What this means is that applications are not limited to the features or subjected to the pitfalls of any single, centralized data store. For example, applications that must search data semantically can use a triple-store. However, triple-stores are terrible for applications that may provide tools to analyze Activity Based Intelligence (ABI), where graph databases are much better solutions. With ~~CubbyDB~~Canon, event handlers can be created to feed both of these and others if desired. This creates an architecture where applications themselves are decoupled from data consistency. Tailoring the data store for specific application retrieval requirements results in much better application performance and faster time to market with the application. Applications designed in this manner are smaller and simpler and these applications may "come and go" without risking the underlying data.

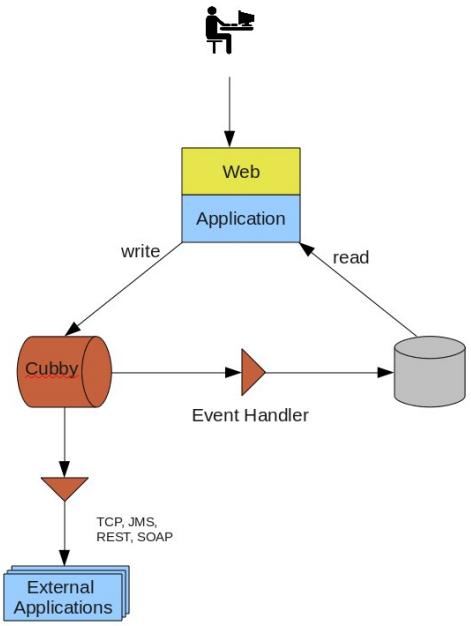

Figure 2: (U) Example ~~Cubby~~Canon Application Architecture

(U) Much of the "magic" of ~~CubbyDB~~Canon comes from the EventHandler. This simple interface allows any subscriber (application) to simply replay events as they were recorded from any point in the event sequence. As events are received by the EventHandler, applications process the data it needs for its specific business requirement and optionally stores the data in the format the application needs for retrieval. This architecture naturally decouples the data consistency requirements from the application, and lets developers focus on the needed business requirement. Data stores can be temporary, starting over becomes an easy option; just replay the events into the new store. Applications can also start or stop reading events at any point in an event sequence, joining or rejoining at any time.

(U) Similarly, applications that change data can continue to do so locally in their own event sequences. ~~CubbyDB~~Canon will merge the event sequence for the distributed applications. ~~CubbyDB~~Canon exposes a second pluggable interface, the EventMediator, which allows business logic to be added to ~~CubbyDB~~Canon nodes to automatically reconcile event collisions. Combined with EventHandlers, applications based on ~~CubbyDB~~Canon are capable of operating over an unreliable or completely disconnected network that allows them to eventually synchronize as they return to a network. Details on ~~CubbyDB~~Canon replication will be covered a related article "Building Distributed Applications with ~~CubbyDB~~Canon".

(U) Use Cases

(U) Implications of this model are vast. As the first article in the series examining the potential use cases for ~~CubbyDB~~Canon, a few use cases are examined at a high level; future articles will go into much greater detail. We'll focus on some of the non-functional requirements that led us to developing ~~CubbyDB~~Canon, and briefly describe how ~~CubbyDB~~Canon meets our needs.

(U) Distributed (WAN) Application Deployment

(U) Unlike applications such as Facebook, Google, LinkedIn, etc. that are often referenced as scaleable and highly available systems, IC systems have unique requirements and constraints which impact the design of a scaleable and highly available system. in the networks are generally less reliable and less resilient and the data is much more critical. For example, IC data is much more critical, and irreplaceable, than data contained by the aforementioned companies, and so a higher level of safety must be designed into the system. Further, any IC system that hopes to provide support to tactical customers must be able to operate both as an enterprise/cloud service and as well as a local, "stand alone" application capable of offline operations.

(U) Despite the dissimilarities, the data consistency problems that IC applications face are similar to Facebook, Google, LinkedIn, etc. Both commercial and IC systems incur data consistency problems stem from the massive amount of data, but IC applications also challenged by the environment where our customers are deployed. Complex networks managed by multiple government agencies, tactical customers over low-bandwidth and unreliable connections, and the complexity of multiple networks and each with different levels of access are just some of the aspects that are often juggled by an application developer. ~~CubbyDB~~Canon aims to alleviate many of these aspects by handing them in a consistent and predictable manner, making the task of developers and security auditors alike much easier.

(U) Storing data as a sequence of immutable events allows ~~CubbyDB~~Canon to provide the data replication and reconciliation features necessary to support WAN distributed multi-write (multi-master) instances of an application. Replication between ~~CubbyDB~~Canon clusters are achieved by defining a relationship between the two clusters, which can be optionally defined with an event filter and one or more EventMediators. The filter ensures that only events that are intended/permitted for the recipient cluster will be sent. EventMediators provide a means to automatically reconcile data collisions between the two clusters. With data replication and reconciliation handled before the data makes it to the application, applications can forgo the need to choose data storage solution(s) based on these non-functional capabilities, and instead focus on choosing a solution which best supports their business needs.

(U) GETS applications use ~~CubbyDB~~Canon to provide data replication and reconciliation between enterprise and tactical application deployments. GETS applications are designed to run on platforms at all levels, from enterprise/cloud deployment down to tactical offline field deployment, which is made

possible with the aid of ~~CubbyDB~~Canon. ~~CubbyDB~~Canon stores the sequence of changes made by the application, allowing for applications which change data to send those changes to remote ~~CubbyDB~~Canon clusters when possible. Offline application can continue to operate independently and synchronize updates once the application is rejoined with its replication partners. While applications are designed to reduce the chance for collisions, some will still occur, and EventMediators at each cluster will automatically handle the collisions based on defined business rules. EventFilters are employed to tailor the data sent to each tactical system, sending only data relevant to the specific tactical users. This model enables GETS to deploy the same application code between platforms, which ~~CubbyDB~~Canon handles the data integrity aspects

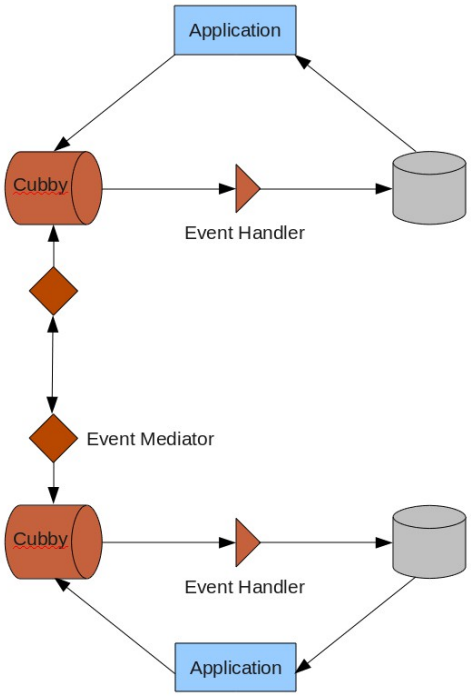

Figure 3: (U) Example ~~Cubby~~Canon Replication Architecture

(U) Seasoning New Applications

(U) As applications evolve, and business requirements change, new business rules are added to an application to meet those requirements. Often, these requirements uniquely process data as it is entered into the application. Applications which store their data as its current state must either deploy the new feature and attempt to process existing data through a unique (one-time use) data integration process, or, if that isn't possible because the data that is stored is not adequate, wait for new data to arrive. Application using events as its definitive data store can instead replay the events through the new code, allowing new application features to process the data as if it had been in place since the start. Similarly, if features are determine to be incorrect (for example, a bug in the logic), the offending code can be redeployed and data can be reprocessed as it was received originally.

(U) ~~CubbyDB~~Canon application design best practices recommend applications be as small as possible, allowing for small units of code to be replaced as necessary. The data those components store can just

as easily be deleted and recreated as needed from the event sequence (see Figure 4). This application design minimizes the overhead necessary in creating and applications while allowing the applications to derive its own significance from the data history.

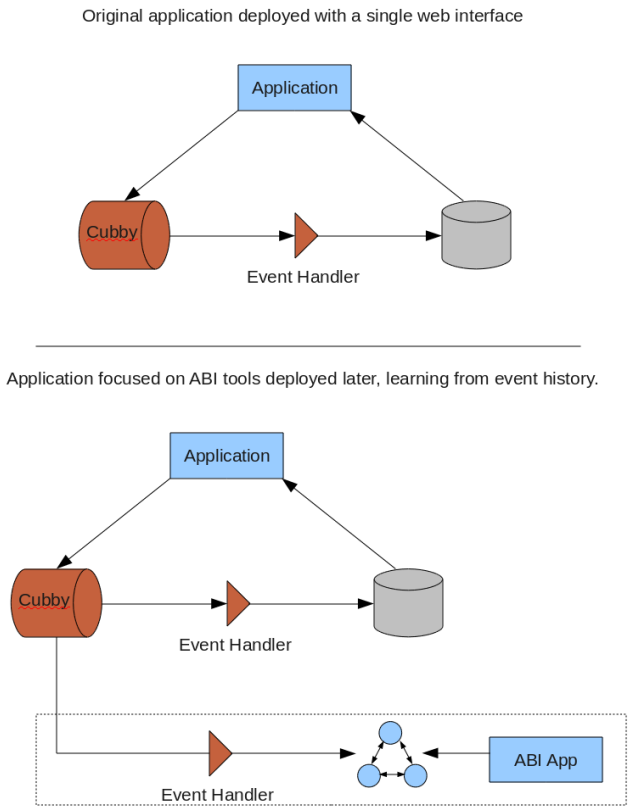

Figure 4: (U) Adding New Application To Existing ~~CubbyDB~~Canon Sequence

(U) Similar to initial loading of data, the ~~CubbyDB~~Canon architecture drastically reduces the cost of an application failure. Reducing the impact of application failure is an interesting aspect to discuss, but the implications are important. No application is perfect, and defects can make it past even the most careful test and release management procedures. A traditional application with write and read access to a database that contains a bug can easily corrupt the data in the database. Hopefully, the database was recently backed up and all the data is recoverable, but this is complicated since some of the data in the database history is actually bad, and many databases do not provide selective recoveries. In contrast, applications that process events and create their own application-specific read store can simply be deleted and redeployed with the bug fix (see Figure 5). In the worst case, where an application writes dangerous data to the ~~CubbyDB~~Canon, ~~CubbyDB~~Canon supports a familiar concept to IC applications – event sanitization. Sanitization supports the deletion of an event within a stream, and ~~CubbyDB~~Canon clients are notified of this deletion, so any downstream copies or changes can be rolled back. Sanitization will be covered in a future article entitled “Sanitizing Events: Recovering from Corrupt or Data Spillages with ~~CubbyDB~~Canon”.

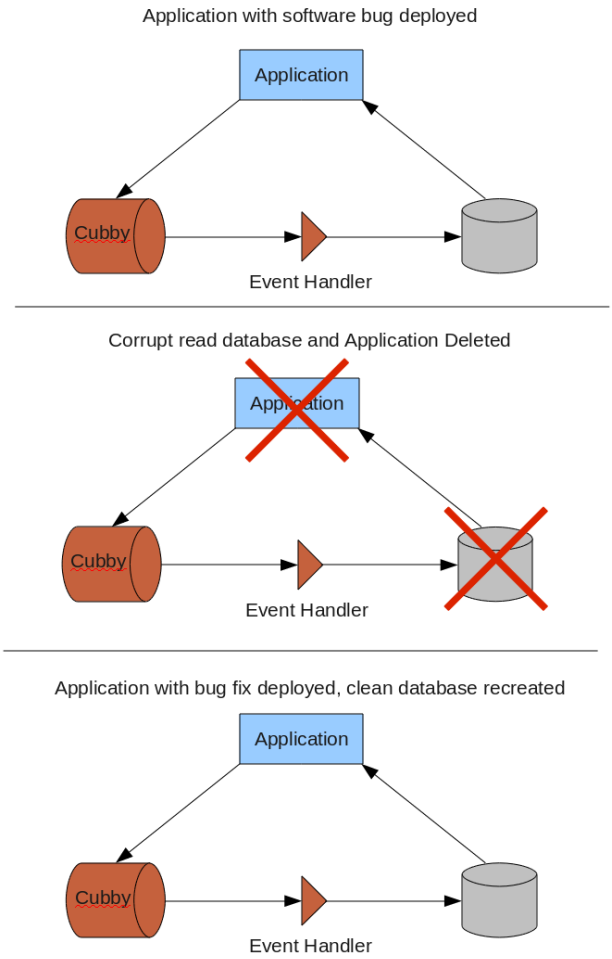

Figure 5: (U) Replacing Corrupt Read Database

(U) Adding Security: Inherited Audit Logs and Application-level IDS

(U) All IC systems must pass an intensive security accreditation process based on requirements from NIST, DISA, and local policies. IA requirements ensure applications meet availability, security and policy requirements as well as follow best practices in application design and management. Security accreditation is a continuous process beginning with initial accreditation and continuing through the life of a system, including software updates and risk discovery. IA requirements and accreditation are currently the largest impediment, risk, and ongoing cost to a software project destined for the Intelligence Community. IA requirements are so extensive that software architects and developers must be well-versed in the requirements prior to designing application code, as they can significantly impact the way an application is designed.

(U) One such IA requirement that requires careful implementation is audit logging. Unlike audit logging found in most applications, which focus mainly around logging specific operations which may be a security concern, I*C* applications are required to log not only changes to system and data, but also who has accessed the data within the system. Further, applications are required to make use of these logs, not only to notify the proper individuals for different concerns, but also to take action on the events themselves. For example, applications must be capable of detecting a data breach and then must

fail secure, preventing further access. Traditional applications might then resort to using Big Data tools and roll their own solution, or purchase 3rd party software to read logs and take action. Any of these solutions create additional overhead for development, deployment, management and auditing of the system.

(U) Audit logging are requirements that must be implemented in each IC application. Further, since auditing is handled within application code, tests must be conducted to ensure that every area of the code does in fact audit actions. It is too easy to miss one operation's auditing, especially in follow-on code releases. By using ~~CubbyDB~~Canon, applications inherit auditing at all levels since every operation is logged as an immutable and attributable event. In this way, ~~CubbyDB~~Canons event sequence storage approach serves double duty as both the definitive data record as well as the audit history of access and changes to the events. Not only is this burden removed from each application leveraging ~~CubbyDB~~Canon, it also dramatically reduces the amount of work necessary on an accreditation – everything must be logged because, if it is not, data does not change. Auditors can rely on the fact that everything is audited, and generic tooling can be used to prove this, rather than having to create unique test procedures for each scenario.

(U) Another IA requirement mitigated simply through the use of ~~CubbyDB~~Canon is the ability to take action on complex sequence of events which indicate an application security breach. For example, there are STIG application checklist items which require applications to detect if a single account is concurrently accessed from two different physical locations. Further, upon detecting this, the application must disable the offending account. Rather than requiring each application to development an implementation to meet this requirement, ~~CubbyDB~~Canon allows this to become an administrator (runtime) concern, handled centrally of any supported application. A separate handler, not implemented in any one application, can be deployed to process events as they occur across multiple applications. This separate handler can then take action if it detects a violation, and can disable a user's account. This model is similar to network appliances known as Intrusion Detection Systems (IDS), which observes the meta and content of packets as it passes a choke-point, and may take passive (logging, notification) or active (disable network segments) actions. Like an IDS, ~~CubbyDB~~Canon EventHandlers can be deployed as an appliance and effectively satisfy requirements for multiple applications simultaneously (see Figure 6). Again, this results in simpler applications focusing on the business problem, and a system that is easier to secure and accredit.

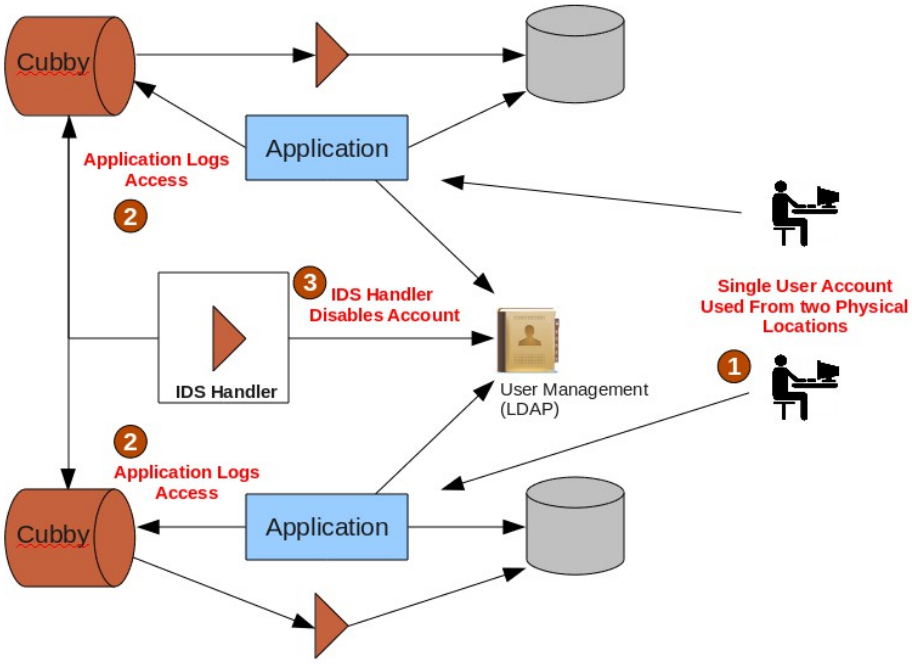

Figure 6: (U) ~~CubbyDB~~Canon Handler as Application-level IDS

(U) Conclusion

(U) ~~CubbyDB~~Canon is a core enabling technology for GETS Version 3. ~~CubbyDB~~Canon was carefully designed to meet the needs of applications deployed within the IC. It also provides capabilities not found in any single NoSQL or event messaging technology. The unique way ~~CubbyDB~~Canon stores this data allows the GETS team to deliver smaller applications and reduce the time required to build them while also making them easier to accredit. ~~CubbyDB~~Canon also allows GETS to meet the needs of war fighters by enabling our applications to run concurrently both in enterprise/cloud environments as well as disconnected systems in theater. While the approach ~~CubbyDB~~Canon takes is not foreign to application developers, it does require applications using ~~CubbyDB~~Canon to support a unique application architecture, which promotes smaller more modular application design, making applications easier to develop, accredit, and manage.