A Resilient and Secure Software Platform and Architecture for Distributed Spacecraft

William R. Otte and Abhishek Dubey and Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University

ABSTRACT

A distributed spacecraft is a cluster of independent satellite modules flying in formation that communicate via ad-hoc wireless networks. This system in space is a cloud platform that facilitates sharing sensors and other computing and communication resources across multiple applications, potentially developed and maintained by different organizations. Effectively, such architecture can realize the functions of monolithic satellites at a reduced cost and with improved adaptivity and robustness.

Openness of these architectures pose special challenges because the distributed software platform has to support applications from different security domains and organizations, and where information flows have to be carefully managed and compartmentalized. If the platform is used as a robust shared resource its management, configuration, and resilience becomes a challenge in itself.

We have designed and prototyped a distributed software platform for such architectures. The core element of the platform is a new operating system whose services were designed to restrict access to the network and the file system, and to enforce resource management constraints for all non-privileged processes Mixed-criticality applications operating at different security labels are deployed and controlled by a privileged management process that is also pre-configuring all information flows. This paper describes the design and objective of this layer.

Keywords: Distributed Real-time systems and embedded systems, Information Flow Controls, Multiple Levels of Security

1. INTRODUCTION

A fractionated spacecraft is a cluster of independent satellite modules flying in formation that communicate via ad-hoc wireless networks. Together these independent satellite modules provide a a cloud platform that facilitates sharing sensors and other computing and communication resources across multiple applications. This architecture can realize the functions of monolithic satellites at a reduced cost and with improved adaptability and robustness.[?] Several existing and future missions use this type of architecture including NASA's Edison Demonstration of SmallSat Networks, TANDEM-X, PROBA-3, and PRISMA from Europe. In each of these missions, the cooperating fractionated satellites are expected to provide the foundations for truly distributed software applications, used by many, possibly concurrent missions.

Possibility of sharing a relatively stable and common distributed platform across applications and missions from different organizations poses unique challenges. Missions will have different security requirements and policies. Data belonging to different organizations might have to be compartmentalized and protected from each other. Additionally, as the platform can host applications supplied by different organizations, each of which may have different security requirements and classifications, it must ensure strict isolation. Furthermore, the platform itself is a critical resource and it must protect itself and as such its must be carefully monitored and controlled. In fact we expect that these systems will always be managed by some authority.

All these requirements imply that *security* cannot be an afterthought. Information flows in general and access to shared resources in particular should be controlled under some overarching policy. Applications should not be able to interact arbitrarily, but rather only if they have compatible security classifications, and even then only if such interaction is explicitly authorized. If multiple applications run on the platform concurrently, and

---

Further author information: Send correspondence to William R. Otte, 1025 16th Ave S Suite 102, Nashville, TN 37212, E-mail: wotte@isis.vanderbilt.edu

there is a need for some degree of data sharing among the applications, the platform must permit that while enforcing the security policies defined for the system. Since the applications themselves may not be fully trusted, the platform itself must provide a secure execution environment that prevents arbitrary flows of information between applications. *Resilience* is also essential. Anything can go wrong at any time. Moreover unanticipated changes in the system (erroneous updates) or in the environment must be survivable and the system should recover.

As an example consider the scenario where two satellite modules exist. One module has a high resolution imaging camera, while the other satellite has a lower resolution camera. The second satellite has additional hardware to allow efficient image compression. Also the second satellite has a direct high bandwidth transmission link to ground. Now consider two different mapping applications running on these satellites. One of them is a highly sensitive application that captures high resolution image, while the other one only captures low resolution image. Both of these applications need to run on the cluster simultaneously, use the image processing hardware, and send the data to the ground. However, at no time should the low resolution application get access to the high resolution application and their data should be isolated from each other even when being transmitted to ground. Management of these applications entails providing for secure information flows, application lifecycle management and strict partitioning of computation resources so that these applications are temporally isolated.

Distributed **RE**altime **M**anaged **S**ystem (DREMS) platform [?] (DREMS) is a full information architecture that enables such systems. The architecture consists of (1) a design-time tool suite for modeling, analysis, synthesis, integration, debugging, testing, and maintenance of application software built from reusable components, and (2) a run-time *software platform* for deploying, executing, and managing application software on a network of mobile nodes. The run-time software platform consists of an operating system kernel, system services and middleware libraries. Distributed applications are composed from reusable components, which are hosted inside 'actors'. Actors specialize the notion of OS processes; they have persistent identity that allows them to be transparently migrated between nodes, and they have strict limits on resources that they can use. Each actor is constructed from one or more reusable components[?] where each component is single-threaded.

The operating system restricts access to the network and the file system, enforces resource management constraints for all non-privileged processes. Mixed-criticality applications operating at different security labels are deployed and controlled by a privileged management process that is also pre-configuring all information flows. Resource constraints are addressed by strict temporal and spatial partitioning and a resource quota system, while access to physical resources, including payload modules, is controlled by multi-level security policies. In prior work, we have described the general architecture of *DREMS* ,[?] its design-time modeling capability,[?] and its component model used to build applications.[?]
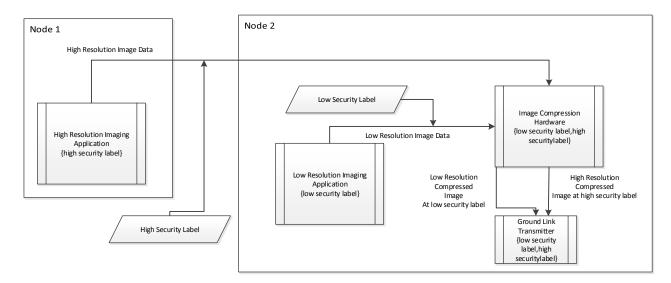

Figure 1. Two Application Example

This paper describes the *Secure Transport (ST)* mechanism built into the DREMS OS. ST is a network transport layer that enforces information flow partitions based on security classifications. ST uses Multi-Level Security (MLS) labels[?] to represent security classification; and therefore, MLS policies are used to enforce information partitioning based on a set of linearly ordered hierarchical classification levels and non-hierarchical need-to-know categories. The standard MLS policy states that information can only flow on the same level or from a lower to higher level according to the dominance relation. In addition to supporting information partitioning based on this MLS policy, ST also restricts any communication topology that can be established between any two application processes without any governance from another process with elevated system privileges. This mechanism is essential to our system since our *threat model* assumes that Application Actors cannot be trusted and therefore they can only interact with each other under certain constraints . We will discuss the ST in detail in subsequent sections.

Figure 1 shows an example of the system configured with strict flows and the security labels. Both applications image data must go through the image compression hardware on node 2, and the application data must remain separate. The high-security high-resolution image application transmits its data which is protected by a high-security label, while the low-resolution image application transmits its data that is protected by a low-security label. Using these labels, the network ensures that no unauthorized data reception or transmission is allowed. All devices are wrapped and managed by a separate actor which then allows other actors to use the services provided by the device. That is, actors cannot arbitrarily use the operating system service to communicate with devices. For example, the image compression hardware is encapsulated in a multi label actor, which has to be developed per high-confidence standards because it has to be trusted with keeping the two information flows isolated. Moreover, the scheduler ensures that the application data served by the hardware remains temporally and spatially isolated through the use of ARINC-653 style partitions (not shown in the figure). Temporal partitioning provides a periodically repeating fixed interval of CPU time that is exclusively assigned to a group of cooperating tasks. However, this style of scheduling does necessarily lend to information separation if some state is left in the device from one group of tasks to other group of tasks. Therefore, the actor managing the device need to scrub the device before switching between information flows if they have different labels.

2. RELATED WORK

The secure transport feature of DREMS is based on multi-level security (MLS).[?] All messages have a security label and must obey a set of mandatory access control (MAC) policies. The main novelty in DREMS with respect to MLS is the concept of multi-domain labels[?] to support secure communication among actors from different organizations. Decentralized label solutions have been described in the literature.[?] However, at present our system implements a centralized label model.

There are various real-time operating system products with sophisticated development toolchains (e.g., Integrity by Green Hills), and there are systems that support Multi-Level Security (e.g., SELinux). However, to the best of our knowledge we are not aware of a run-time platform that holistically provides the capabilities required to support a dynamic yet compartmentalized distributed platform where applications from different security domains and organizations can be managed and updated over time.

3. SECURE TRANSPORT

An essential requirement of any distributed application is that its constituent components, which may be distributed across several nodes and address spaces must be able to communicate. While this is well accomplished in traditional distributed applications using derivatives of the Berkeley Sockets API that are common across almost all platforms, these traditional networking features and services do not fully meet the requirements that *DREMS* is intended to address.

In order to support communication and coordination between applications of different criticality, priority, and security levels, we developed the DREMS Secure Transport (ST) facility. ST is a managed communications infrastructure that provides for datagram oriented exchange of messages between application tasks. ST restricts the transmission of datagrams according to both a communication ([?,?,?]) topology and a Multi-Level Security (MLS) policy, both of which must be configured for each task by a trusted system administration infrastructure.

MLS defines a policy based on partially ordered security labels that assign classification and need-to-know categories to all information that may pass across process boundaries. This policy requires that information only be allowed to flow from a producer to a consumer if and only if the label of the consumer is greater than or equal to that of the producer. Note that ST facility is part of the system's Trusted Computing Base (TCB).

This communication may take place at several different scopes: First, two components that are running on different nodes must be able to securely exchange messages over the network. Second, two components may be running on the same node; while this could be accomplished as with components on different nodes, the collocation of these processes allows for optimizations in delivery, as use of the operating system network stack is not required. Third, each process may have multiple threads of control that need to communicate amongst each other. Finally, each application process may require the ability to communicate with resources provided by the kernel. Such communications might involve, for example, the kernel notifying the application of timer expiration or communication with device drivers.

This section will provide an overview of the core concepts of DREMS Secure Transport and discuss the architecture of its implementation.

3.1 Endpoints

Endpoints are the basic communication artifact used by applications to transmit and receive messages; they are analogous to socket handles in traditional BSD socket APIs. Like traditional sockets, user space programs pass an endpoint identifier to the `send` and `receive` system calls. Unlike traditional sockets, however, unprivileged tasks may not arbitrarily construct endpoints that allow for inter-process communication with other tasks; such endpoints must be explicitly configured by a privileged task acting as a trusted system configuration infrastructure.

Endpoints are separated into four different categories with different restrictions on their creation and use:

- **Loopback Endpoints (LBE)**: Loopback endpoints may be created on demand by a task and are used to communicate amongst threads of that task, but may not be used to communicate with other tasks. These endpoints are not subject to restricted communication based on flows or security rules.

- **Local Message Endpoints (LME)**: Local message endpoints are the basic method of IPC, and may be used to send messages to other tasks hosted by the same operating system instance. These endpoints must be configured by trusted system configuration infrastructure and are subject to restrictions placed by flows and security rules.

- **Remote Message Endpoints (RME)**: Similar to LMEs, RMEs are a mechanism for IPC between tasks, but may be used to communicate with tasks hosted by different operating system instances through a network.

- **Kernel Message Endpoints (KME)**: These endpoints are used for communication with the operating system kernel. These endpoints may be created by an unprivileged task and used to register for important system notifications, or may be associated by a privileged task for communication with device drivers.

3.1.1 Addressing

In traditional networking APIs, addressing is accomplished with an IP address and port pair that implicitly defines a destination task by virtue of the fact that a task can request to bind to that particular port when it starts on a node owning that particular IP address. This makes the architecture brittle in case the process has to be migrated.

In ST, however, each actor is assigned a globally unique identifier, analogous to a process identifier in traditional operating systems. This global identifier, when combined with a numeric endpoint identifier, forms a unique address for a specific port in a specific actor, regardless of its location in the distributed system. Associations between this actor identifier and a particular node/IP address are made by privileged system configuration infrastructure, and is not exposed to regular applications. This way, the actual information flow association between two actors does not have to be reconfigured if the actor is migrated from one node to the other.

3.2 Supporting Administrative Control Over Inter-Process Communication

The ST infrastructure restricts the ability of processes to spontaneously communicate with other processes in the system; reserving this configuration capability to trusted deployment and configuration infrastructure that is part of the TCB. This is a desirable property for safety- and security-critical systems: a badly behaved or malicious application component, given unrestrained access to then network, could probe to discover information about applications they are unauthorized to know the existence of, or worse, actively disrupt the functionality of other applications. Moreover, in the context of policy described later in Section 3.3, an application should not be allowed to set labels on itself or its communication endpoints. This administrative control is achieved by using flows. Flows are configured by a privileged user level service and facilitated by the kernel, both of whom are part of the Trusted Computing Base (TCB).

3.2.1 Flows

In ST, communication is allowed between two LME or RME endpoints if and only if there exist mutually compatible *flows* on each endpoint. A flow, assigned to an endpoint, is a connectionless association with an endpoint owned by a designated process (in *DREMS* , process identifiers are statically assigned and globally unique). This association determines if the local endpoint is allowed to send or receive messages with the remote endpoint.

- **Source Flow**: Indicates that the endpoint that owns the flow is allowed to send messages to the identified recipient.

- **Destination Flow**: Indicates that the endpoint that owns the flow may receive messages from the identified recipient.

- **Bidirectional Flow**: Indicates that messages may be both sent to and received from the identified recipient.

In all cases, flow assignment between two endpoints must be mutual in order for communication to succeed, *i.e.* a sender must have an outbound flow to the recipient, and the recipient must have a inbound flow from the sender.

3.2.2 Endpoint and Flow Management and Creation

The system calls provided by the ST infrastructure are bifurcated into two groups: privileged and regular. The privileged interfaces may only be used by privileged platform actors that are part of the TCB. These interfaces are used to create and manage the configuration of endpoints and flows for the RME, LME, and KME endpoint classes; regular applications are not allowed to manage their own configuration. The regular system calls may be used by any Actor to retrieve handles to endpoints created by system manager, gather flow configuration information, create LBE endpoints, and to send and receive messages.

3.3 Mandatory Access Control Policies

In order to meet the security objectives of the system, ST must enforce Mandatory Access Controls (MAC) on all information flows. In this system, every computing node has a set of security labels that restrict the classification and category of information that may be sent/received by that node. Each process, in turn, has a subset of the node labels which it is allowed to send/receive. Finally, each communication endpoint has a subset of the labels held by the owning process. The application must assign a label that is compatible with the communication endpoint it using to send the message. The operating system kernel, part of the TCB, will ensure that an application is not allowed to send messages with labels that it does not own. Similarly, before delivering a message to an actor, the kernel will verify that the recipient has the label attached to the message to be delivered.

The ST MAC implementation supports, obviously, the basic case where communications are allowed between parties that have equivalent labels. In addition, certain communications between parties of unequal labels are also allowed: *read down*, where a process with a higher label is allowed to receive a message from a process with a lower label, and its companion *write up*, where a process with a lower label is allowed to send a message to a process with a higher label.

3.4 Supporting a Variety of Quality of Service Properties

Modern transport protocols and hardware support a number of properties that influence the quality of service experienced by application messages. This includes two factors that are readily apparent: protocol selection and device/hardware level support. For example, the communications hardware may support additional QoS parameters like priorities that influence the delivery of messages over that medium. Examples may include priorities, time triggered traffic classes, or bandwidth reservations.

As an application level protocol in the OSI model, ST may use any number of transport protocols to transmit messages. Each transport level protocol, may provide a number of useful features that are intrinsic to its design; examples of such properties include reliability, ordering of delivery, congestion control, and differing connection and multiplexing properties. For applications that do not require reliability or ordering, UDP is appropriate. For applications that require both reliability and ordered message delivery, SCTP may be appropriate.

3.5 Supporting Cryptographically Secure Communications

Finally, in order to fully meet the security objectives of the system, all communications through ST are cryptographically secure. In particular, ST must ensure must ensure that the following properties are protected for all messages: confidentiality, ensuring that messages cannot be observed by a third party; integrity, ensuring that messages are not altered while in transit; and authenticity, ensuring that message authorship cannot be forged. In order to support these properties, well known transport level cryptographic schemes such as IPSec are used. Additionally, the system should also provide a cryptographically secured physical layer protocol. Note that the IPSec support in Secure Transport is a work in progress. Also, the implementation of an encrypted physical medium is out of the scope of ST implementation.

4. SECURE TRANSPORT IMPLEMENTATION

The implementation of Secure Transport in DREMS is broadly divided into two halves, which we will call the "ST Support" and the "ST Reactor". The Support portion of the ST implementation contains most of the support infrastructure, *i.e.* bookkeeping data and business logic required to implement basic operation, including enforcement of flows and security restrictions. This portion also serves as the primary point of entry for all system calls related to ST. The Reactor contains the data structures and business logic required to send and receive remote messages using the underlying UDP and SCTP protocol stacks.

4.1 Secure Transport Support

Access to ST Support is controlled by a single read/write mutex preventing race conditions that may occur when a privileged task attempts to update the system endpoint/flow configuration while regular tasks are performing unprivileged operations. This read/write mutex has three important properties:

1. It ensures that when a write lock is requested, no further read locks are granted; ensuring that a pending write (typically by a high priority task) will not be indefinitely blocked.

2. It allows multiple readers to acquire the lock at once, reducing the possibility of a priority inversion blocking access to high priority tasks.

3. It is a *sleeping lock*, meaning that if the mutex cannot be acquired by a task, it is marked inactive and not scheduled again until either a timeout occurs or the lock can be obtained.

4. It prevents *priority inversion* by supporting priority inheritance: when a high priority process is blocked while trying to acquire the mutex, the priority of the current owner(s) is boosted to that of the top priority waiter until they release the mutex.
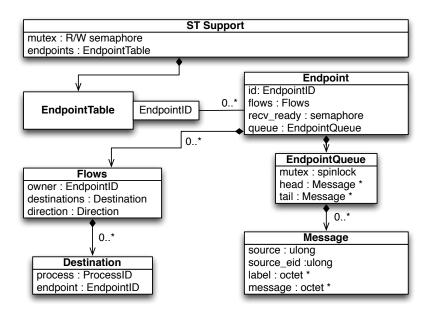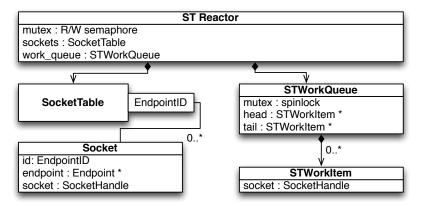
Figure 2. Simplified ST Support Architecture

Figure 3. Simplified ST Reactor Architecture

A *write* lock is obtained whenever a system call is entered that would create, update, or modify an endpoint - such system calls may only be used by high priority privileged tasks. Regular system calls — *e.g.* send, receive, or select — acquire the mutex with a *read* lock only. This allows multiple tasks of different criticality and priority levels to be active in the ST infrastructure at once.

Most data structures are satisfactorily protected using this scheme: all data structures (for example, tables maintaining endpoint configuration, flow configuration, and destination mappings) with one exception (discussed below) are read-only to tasks performing unprivileged ST operations.

For this reason, most of the design of the ST Support is outside the scope of this paper. The data structures that maintain state for endpoints, shown in Figure 2, however, merit further discussion. Endpoints are represented in the kernel by a data structure that contains:

- Basic properties of the endpoint, *e.g.* the identifier, type, security label, and maximum allowed buffer sizes.

- The endpoint queue, a linked list of message objects waiting to be received.

- A spinlock used to synchronize access to the endpoint queue.

- A mutex (sleeping lock) used to wait for receipt of messages.

The basic properties of the endpoint are not modified by tasks accessing the endpoint through non-privileged system calls, and are sufficiently protected by the ST read-write mutex described above. The endpoint queue, however, is expected to be manipulated by these non-privileged system calls, and if not properly protected could be a source of priority inversions and deadlocks. In order to avoid these cases, the endpoint operations use a spinlock - a low overhead busy wait mutex that temporarily disables preemption while in its critical section, to protect access to the endpoint queue. In the event that a task attempts to receive a message from an empty queue, a mutex (sleeping lock) is available so the task can yield to lower priority tasks that may be attempting to send messages into that queue. In order to illustrate this, we will describe the process through which messages are sent and received for LMEs.

Local Message Endpoint - Sending a Message   When a task enters the *send* system call, it first acquires a read lock on the ST mutex. After successfully acquiring the mutex, it retrieves the endpoint structure for the sending endpoint and checks for a valid outbound flow for the destination. Presuming a valid flow is found, it then retrieves the endpoint structure for the destination endpoint, and checks that a valid destination flow exists. If both flow checks pass, the task constructs a message object to hold the message. The task then acquires the destination endpoint's spinlock for only as long as required to place the message object on the endpoint queue's linked list, and signals the endpoint mutex before releasing the spinlock. Note that when the spinlock is acquired, the task cannot be preempted until the spinlock is released.

Local Message Endpoint - Receiving a Message   When a task enters the *receive* system call, it first acquires a read lock on the ST mutex. After successfully acquiring the mutex, it retrieves the endpoint structure for the receiving endpoint. The task acquires the spinlock on this endpoint structure and attempts to remove a message from the linked list. If no message exists, it releases the spinlock and performs a timed sleep on the mutex. Once awakened (either through a signal from a sender or through timeout), it re-acquires the spinlock and attempts again to dequeue a message.

Synchronization Analysis   In the tasks outlined above, two tasks of different priority levels can execute most of the tasks of sending or receiving a message without contending for a mutex. Most of the computationally intensive work required for sending or receiving a message happens while holding only the ST R/W mutex and executes at their respective priority levels. The only place where such tasks may contend is when inserting or removing an item in a linked list, a very fast O(1) operation. Since that preemption is disabled once the lock is acquired, we ensure that a higher priority task cannot preempt this operation and prevent it from completing.
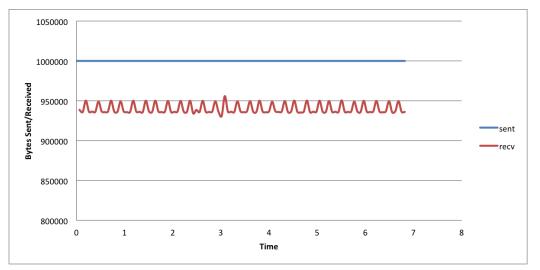
4.2 Secure Transport Reactor

The Reactor portion of the ST architecture, a simplified version of which is shown in Figure 3, consists of the data structures and business logic required to send messages to tasks on other nodes. Similar to the ST Support, the ST Reactor has a read/write mutex to synchronize access to its internal data structures. Unlike the ST Support, however, there is no requirement for per-endpoint synchronization. Also unlike the ST Support the ST Reactor is not entirely driven by threads from user level tasks. The ST Reactor maintains a pool of kernel threads that handle receipt of messages from the network stack and deliver messages to their destination endpoint queues. To illustrate this, we will describe sending and receiving a message on RMEs.

Remote Message Endpoint - Sending a Message   Sending a message on RME is similar to sending a message on a LME until the sending side flow check has succeeded. At that point the task constructs the message object and enters the ST Reactor. Then the task acquires a read lock on the ST Reactor mutex and retrieves the necessary data structure (*e.g.* socket handle) needed to send a message, using non-blocking semantics, to the desired destination address. Finally the task releases both the Support and Reactor mutexes and attempts to send the message.

Remote Message Endpoint - Receiving a Message When a message arrives from the network stack, the ST Reactor is notified via callback that a message has arrived. When this occurs, an item containing the socket handle is enqueued on a work queue serviced by the ST Reactor thread pool. When one of the members of this thread pool accepts the work item, it allocates a buffer and retrieves the data from the network stack. This buffer is then immediately delivered onto the endpoint queue of the recipient endpoint (requiring only that the reactor thread hold the queue-specific spinlock) and wakes any processes that may be blocked on `receive` or `select`. When a process elects to receive the message, the message is decoded and flows and security labels are checked in the context of that process, using its default priority.

Synchronization Analysis Tasks sending messages need only contend for read locks on both the Support and Reactor portions of the ST infrastructure. Therefore such tasks will not interfere with other higher priority tasks trying to send or receive. Receipt of messages by the Reactor kernel threads merits some further discussion: the priority and scope of these threads must be carefully chosen so as to not interfere with the operation of any critical tasks. Several tasks must be accomplished to correctly deliver messages: buffers must be allocated to hold the message content, metadata attached to the message must be demarshalled, finally flow and label checks must be conducted. Performing all of these tasks early (*i.e.* in the scope of the reactor thread pool) ensures that kernel memory is not wasted on messages that might fail any one of these stages and not be delivered. If these threads are scheduled as real-time tasks, high levels of network traffic, *e.g.* a denial of service attack, would starve any tasks of lower priority. For this reason, we have elected to defer most of these activities to actual receipt of a message by a process: the reactor thread simply allocates a buffer to hold the message and enqueues it on the appropriate endpoint queue. Thus, it is safer to run these threads at maximum priority, and the computationally intensive parts of message delivery run at the actual process priority when they are received. Please note that this is our working hypothesis and has yet to be experimentally validated.

5. EMPIRICAL EVALUATION

Figure 4. ST UDP Utilization

To evaluate the performance of the *DREMS* Secure Transport, a multi-computing node experiment was created on a cluster of fanless computing nodes with a 1.6 GHz Atom N270 processor and 1 GB of RAM each. On these nodes, a simple application was deployed, configured with appropriate endpoints and flows to enable bidirectional communication, using UDP as the underlying transport protocol. On one node, a "sender" process was configured to send data at a rate of 1000000 bytes per second; on the second node, a receiver process noted the rate at which it received data. A graph of the bytes sent vs. received is shown in Figure 4. These results demonstrate a utilization of approximately 93 percent.

6. CONCLUDING REMARKS

Distributed and fractionated spacecraft provide the scientific and research community the ability to deploy and run multiple missions and applications on the same hardware. This definitely pose some interesting challenges regarding the required data isolation between applications belonging to different organization.

In this paper we described the Secure Transport (ST) facility in *DREMS* . The ST is a network transport layer that enforces information flow partitions based on security classifications. ST uses Multi-Level Security (MLS) labels[?] to represent security classification; and therefore, MLS policies are used to enforce information partitioning based on a set of linearly ordered hierarchical classification levels and non-hierarchical need-to-know categories. Our current implementation of ST prescribes a centralized but multi-domain label system. In future, we will investigate the necessary extensions required to support a fully decentralized label model.[?]

ACKNOWLEDGMENTS