

# Retrofitting Communication Security into a Publish/Subscribe Middleware Platform

Carlos Salazar and Eugene Y. Vasserman<sup>(✉)</sup>

Kansas State University, Manhattan, USA  
{[csalazar](mailto:csalazar@ksu.edu),[eyv](mailto:eyv@ksu.edu)}@ksu.edu

**Abstract.** The Medical Device Coordination Framework (MDCF) is an open source middleware package for interoperable medical devices, designed to support the emerging Integrated Clinical Environment (ICE) interoperability standard. As in any open system, medical devices connected to the MDCF or other ICE-like network should be authenticated to defend the system against malicious, dangerous, or otherwise unauthorized devices. In this paper, we describe the creation and integration of a pluggable, flexible authentication system into the almost 18,000 lines of MDCF codebase, and evaluate the performance of proof-of-concept device authentication providers. The framework is sufficiently expressive to support arbitrary modules implementing arbitrary authentication protocols using arbitrarily many rounds of communication. In contrast with the expected costs in securing nontrivial systems, often involving major architectural changes and significant degradation of system performance, our solution requires the addition of just over 1,000 lines of code ( $\sim 5.56\%$ ), and incurs performance overhead only from the authentication protocols themselves, rather than from the framework.

## 1 Introduction

Medical devices have a history of being stand-alone units [1, 2], and most devices currently used in clinical environments stay true to this paradigm. Even when a device manufacturer has implemented some interoperability features, they are not designed to interoperate with devices or software from other manufacturers. Interoperability is confined to vertically integrated systems, preventing technology diversification and promoting vendor lock-in. When implemented, connectivity is typically only used for logging device data [1]. Simply put, medical devices do not play well with others. This stands in contrast to other domains such as avionics, which implement cross-vendor interoperability using an integrated platform [3].

Many in the clinical and medical device community see a need for an integrated “system of systems” for medical devices. This has led to the creation of the Integrated Clinical Environment (ICE) standard [4] and the Medical Device Coordination Framework (MDCF) project [5]. MDCF is a publish/subscribe middleware for coordinating medical devices, architected in logical units analogous to those described in the ICE standard.

While interoperable medical systems can provide numerous benefits, such as improved patient safety, reduced medical errors, and automated clinical workflows [5,6], there are serious security and privacy concerns given the sensitive nature of patients’ medical data. An attacker who could alter data or prevent its transmission could cause serious harm, or even death. Consequently, authentication, encryption, and more advanced data protection features must be incorporated into the MDCF. These functions should be implemented in a modular manner, allowing device manufacturers to implement as many or as few features as they want (or can support, given power constraints). Ideally, pre-built standard authentication and encryption modules will be available from a certification body or third-party software developers [2]. The MDCF should maximize compatibility by offering many security implementations, and should be extensible to ease future integration of evolving technologies. Our modular implementation approach is similar to that of Java or OpenSSL security services [7,8], instantiated by name rather than a function call to a specific method.

Taken together, our modifications to MDCF lay the foundation not only for adding robust authentication and encryption capabilities, but also for easing medical device developer workload by removing the need to write authentication and encryption modules from scratch – pre-defined client-side modules can be used with little or no modification, guaranteeing compatibility as MDCF-side module counterparts have already been implemented.

## 1.1 Requirements

The purpose of the MDCF device security framework is to serve as an abstraction layer which allows developers to implement different protocols (modules) for device authentication and data confidentiality without having to modify the framework itself. Building such a framework therefore requires some foresight into which MDCF components need to be modified, and how to design the authentication API to be developer-friendly or mostly transparent. Furthermore, we must temporarily maintain backward compatibility (with older devices which do not implement security) as not to break test cases for other ongoing development efforts which use the MDCF.

The MDCF device security framework must hook in to the existing MDCF code base while maintaining backwards compatibility with existing devices. Integration of the device security framework should not require significantly changing the fundamental design/architecture of the MDCF connection state machine or otherwise disturb the overall logical separation of MDCF components – the incorporation of communication security should be mostly or completely transparent to developers working with (or modifying) the MDCF code or message transport layer.

Our target “users” are developers working on new MDCF-compliant devices. They will interact with this framework in two ways: using security providers on devices they create, and/or by creating security providers (**the API should be expressive, powerful, and easy to use**). We will take authentication providers as our example, but data confidentiality (encryption) providers are

supported as well. There are multiple authentication providers already implemented and provided for developers. However, we do not prevent developers from building their own authentication modules (**the framework should allow developers to implement arbitrary authentication protocols**), as long as the MDCF can support the protocol (**implementing MDCF authentication modules should not require the alteration of the MDCF**). Finally, we must ensure minimal overhead from the authentication framework (**the only source of overhead should be the authentication modules themselves**, and not the hooks and additional calls to the framework core or messaging layer). In other words, if no authentication happens, no security framework overhead should be visible.

## 1.2 Authentication Hooks

The MDCF had no implemented security controls when we began, nor was the software written with security in mind. We modified the MDCF to place security “hooks” in several key places in the code to allow for later implementation and deployment of self-contained security modules (similar in spirit to SELinux [9]). The goal was to ensure that these hooks are positioned in a way that allows maintainers to write security modules that are sufficiently “expressive.” The MDCF communicates with devices over logical channels, so a natural design choice is to “wrap” the channel in a manner that is transparent to higher-level functions, produce correct output when accessed by an authorized entity, and refuse access to unauthorized users or code.

The resulting modifications to the almost 19,000 lines of the MDCF code base are relatively compact: the security framework (without providers) consists of just over 1,000 lines of code (about 5.56% of the total MDCF). We tested the expressive power of the newly-implemented hooks by first developing a NULL authentication provider, similar to the IPsec NULL encryption method [10], then implementing several other modules, including TLS and DSA. This shows that the security framework is sufficiently flexible to implement almost arbitrary protocols as authentication modules, with an arbitrary number of messages exchanged, all transparent to developers unless they do not use the built-in modules but rather choose to implement their own authentication providers.

## 1.3 Authentication Providers

Hooks are only part of the solution implementing device authentication in the MDCF. We must also have a component that encapsulates the actual authentication protocol – the authentication provider. All providers implement a common interface so that providers for different protocols can easily be “hooked up.” Providers come in pairs – one for the MDCF and another for any device which will support this authentication type, to remove the burden of implementing security-sensitive code from device developers. The provider is responsible for reserving channels to communicate, as well as the actual reception and transmission of messages for its protocol. The device version of the authentication

provider, in addition to running the authentication protocol, is also responsible for generating the contents of **authentication** messages sent from a device to the MDCF at the beginning of the connection process. This object identifies the protocol that the device requests to use to authenticate itself. The authentication protocol(s) supported by a device is specified in its metadata, then instantiated by name upon device connection (at runtime). Similarly, the MDCF retrieves its provider by name, based on the protocol specified in a message from the device to the MDCF at device connection time.

## 1.4 Robustness and Resource Allocation

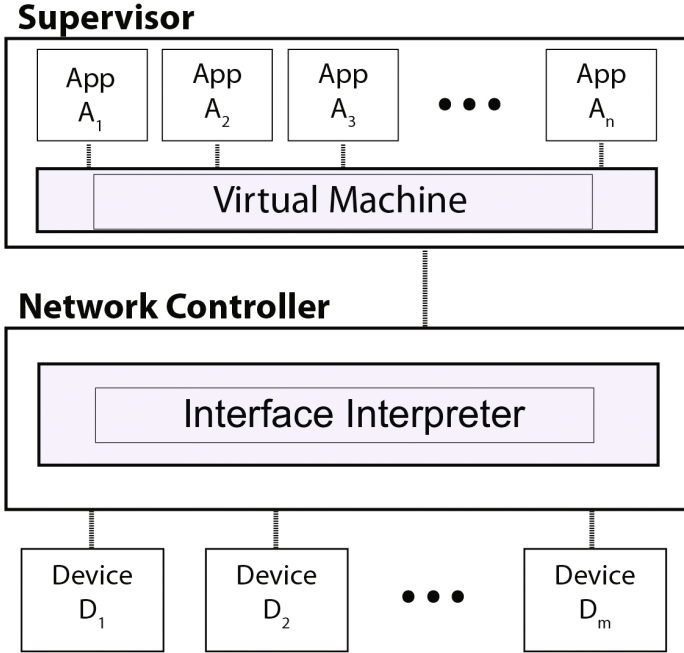
When implementing authentication protocols, it is important to consider denial of service (DoS) attacks in the form of resource consumption (e.g. SYN flood attack commonly used against web servers [11]). To increase robustness of the MDCF to such attacks, we use the laziest practical resource allocation strategy. For example, one potential first step in device authentication is, upon device connection, to create private channels for this device to communicate with the MDCF. These channels are logical addresses used by the underlying Channel Service, also referred to as the message bus (e.g. OpenJMS, ActiveMQ [12]). Note that this occurs before the device has authenticated successfully, and therefore devices which may never be allowed to communicate with the MDCF can tie up resources either through malice or implementation mistakes. Using only pre-allocated resources (specifically, a pool of pre-allocated authentication providers) until after successful authentication allows us to avoid this problem, so we take special care in placing authentication hooks to minimize resource usage. Malicious devices may keep the pool drained, but honest providers should still be able to eventually connect successfully, with wait times bounded in practice with high probability [13].

# 2 Background

## 2.1 ICE

The Integrated Clinical Environment (ICE) is a platform meant to be a ubiquitous standard for medical device interoperability, akin to USB or Wi-Fi in the consumer realm. The goal is to create a functioning system by taking a component-wise approach [2]. In ICE architectures, devices are connected to a component called the Network Controller. This component can be considered the network abstraction: it facilitates communication between devices and applications (automated medical workflows) running on the Supervisor. Figure 1 illustrates the basic architecture of ICE.

**The ICE Supervisor** hosts apps in isolated environments and guarantees runtime resources like RAM and CPU time. In the ICE architecture, apps are programs that can display patient data as well provide control over devices which support it.



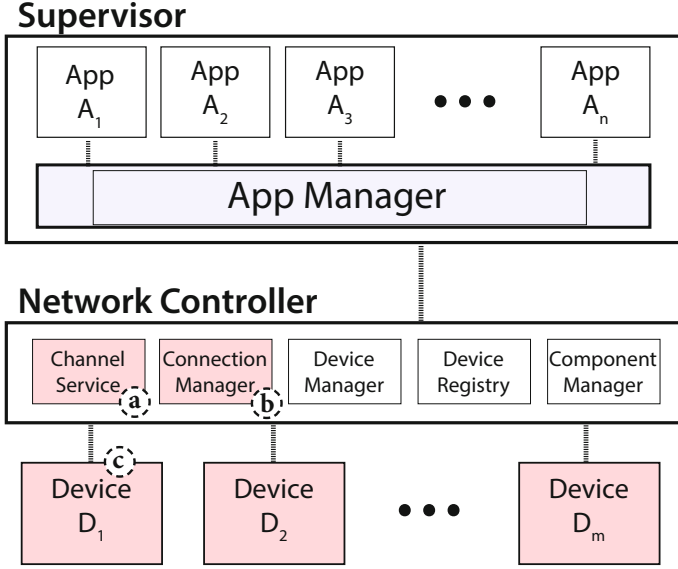
**Fig. 1.** The primary components of the ICE architecture. External interface, and patient connected to devices, are omitted.

**The ICE Network Controller** facilitates communication between Supervisor apps and medical devices. All ICE communication takes place through messages sent over virtual channels maintained by the Network Controller (NC). Each channel is specific to a device-app pair. Whenever a new device connects to an ICE system, it is the Network Controller that “discovers” the new device and performs the connection/handshaking and authentication protocols.

## 2.2 MDCF

The Medical Device Coordination Framework (MDCF) is an open source platform for coordinating and integrating medical devices in order to streamline and automate clinical workflows [5]. The MDCF is intended to be a significant step towards creating a system compliant with the ICE standard. Figure 2 shows the organization of MDCF components with respect to the ICE architecture described in the previous section. They are described in detail below.

**Supervisor Components.** The **App Manager** manages the life-cycle of apps; meaning that it starts and stops the execution of apps, provides isolation and service guarantees, monitors and resolves (or notifies clinicians of) “clinically important” (e.g. medically adverse) interactions or architectural interactions (e.g. two apps trying to get exclusive control of one device).



**Fig. 2.** MDCF components grouped by their logical ICE architecture role and showing primary hook locations (circles a,b,c).

**The Clinician Service** provides an interface for selecting, instantiating, and configuring Supervisor apps for use with a clinician console GUI. New apps can be started and running ones can be configured. Appropriate user authentication/login will be required.

**The Administration Service** provides controls for managing and installing Supervisor apps and components. Appropriate user authentication/login will be required. This service should not need to reconfigure running applications, and should be prevented from doing so by the App Manager.

**Network Controller Components.** **The Channel Service** provides interfaces between middleware platforms and the rest of the MDCF. It contains interfaces for the messaging server (e.g. OpenJMS, ActiveMQ [12]), message senders, message receivers and message listeners. It is partially responsible for inter-app and inter-device data isolation and performance guarantees. It houses the code for all authentication providers, as well as all interfaces and factories related to authentication providers (Fig. 2(a)).

**The Connection Manager** manages connections with devices and the creation and destruction of channels through direct interaction with the message provider. The Connection Manager is directly involved with device authentication. It also contains the main hooks for the Network Controller authentication providers (Fig. 2(b)).

**The Device Manager** sets the status of a device as connected or disconnected, sends commands to devices to start or stop publishing, and configures devices for use with specific apps.

**The Device Registry** stores and retrieves information about devices from a database. For each device, it stores and provides access to information such as its type, name, metadata, and active apps associated with it. We augment this data structure to store security metadata, such as active encryption keys for device private channels.

**The Component Manager** manages MDCF app components and works in a way analogous to the Device Registry. It is used to store and retrieve information about app components.

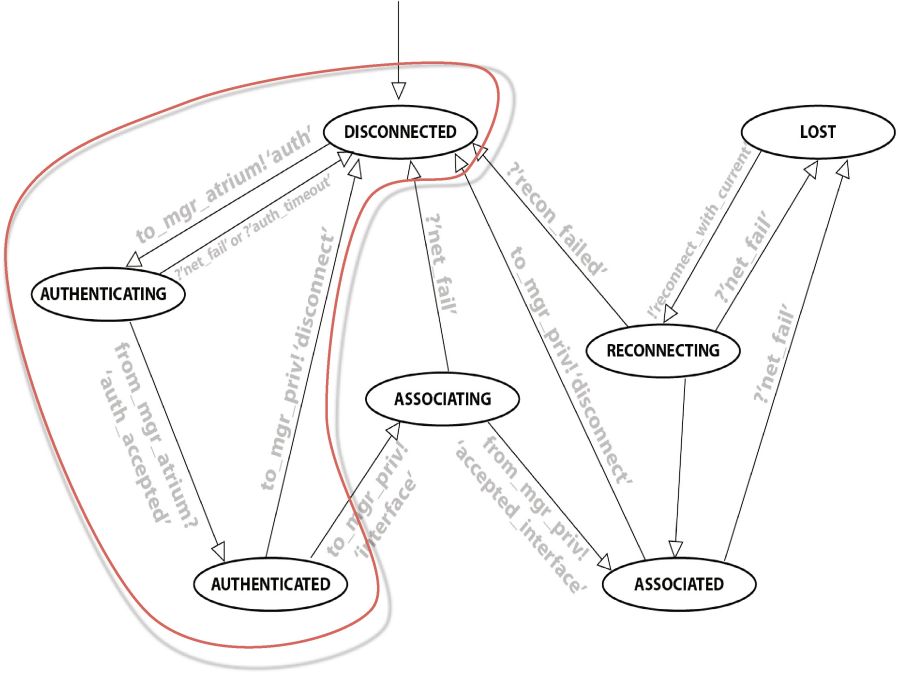
### 2.3 MDCF Connection State Machine

The implementation details of the device connection protocol (using a state machine), shown in Fig. 3, are particularly relevant for device authentication. Each state is implemented as a separate Java class. It is within these classes that the messages in the connection process are sent and received. Effectively, two connection state machines exist for each device; the device and the Network Controller each maintain their own separate views of this state machine. These different views of the connection state machines are utilized to ensure that all of the steps in connection process are executed in the appropriate order. (In the text, **we refer to the Network Controller view** of the state machine, unless stated otherwise.) Each state machine is associated with a single object that can be used to access or modify the current state. These classes are also used to store any information that needs to be accessed by more than one state. The states most relevant to device authentication are:

**DISCONNECTED:** The initial state. The device sends the AUTH message during the DISCONNECTED state. Upon reception of the AUTH message, the Network Controller initializes its view of the connection state machine and moves into the AUTHENTICATING state.

**AUTHENTICATING:** Upon receipt of the AUTH message, the Network Controller allocates and connects to private channels for the device and sends the channel information to the device in an AUTH\_ACCEPTED message. The device connects to the channels, after this point the rest of the messages used for connection are sent across these private channels. Note that “private” is used here not to denote confidentiality, but rather than these channels are logically dedicated to communication with a specific device (as opposed to the public “atrium” channel).

**AUTHENTICATED:** The device has been successfully authenticated. It sends an INTERFACE message to test the private channels before progressing into the ASSOCIATING state. (The INTERFACE message is a confirmation that the private channels set up at the end of the AUTHENTICATING state are working. The content of the message is a fixed string.) Although we routinely



**Fig. 3.** MDCF Connection State machine, with the outline denoting portions relevant to connection-time authentication.

refer to the device authenticating to the Network Controller, it is trivial to extend the protocol to support mutual authentication.

**ASSOCIATING:** Upon receipt of the INTERFACE message, the Network Controller creates heartbeat and acknowledgment channels. The device periodically publishes heartbeat messages on this channel, enabling the Network Controller to detect an unexpected device disconnection (if too many heartbeat messages are missing). The Network Controller communicates these private heartbeat channels to the device and then transitions to the ASSOCIATED state.

**ASSOCIATED:** The device is fully connected in this state. The device will remain in this state unless it ceases to be connected and transitions to either the LOST or DISCONNECTED state.

**LOST:** When too many device heartbeats have been lost, the Network Controller places the device in the LOST state. It must then attempt to reconnect, transitioning into the RECONNECTING state. (If the device state machine is not in the LOST state, and it successfully communicated with the Network Controller, it will be explicitly told to reconnect.)

**RECONNECTING:** A device in this state is attempting to reconnect. If reconnection is successful, the device it returns to the ASSOCIATED state. Otherwise, the device transitions to the DISCONNECTED state. The Network Controller remains in the RECONNECTING state for a fixed amount of time, or until the device successfully reconnects. Only devices which have been previously authenticated may be in the reconnecting state. Depending on the specific authentication protocol used, device credentials may become “stale” while it is attempting to reconnect, and it will be moved to the DISCONNECTED state.

In practice, to minimize resource usage and protect against resource DoS attacks, the state machine object used for a connection by the Network Controller (in the transition from DISCONNECTED to AUTHENTICATING) is taken from a pool of pre-allocated objects. When the Network Controller “destroys” a state machine, it is returned to the object pool. Note that this prevents devices from connecting when the pool is exhausted (either due to a large number of connected devices or an active attack). This is by design: devices that authenticate successfully will eventually connect, and devices which cannot authenticate but are performing the attack cannot cause more objects to be allocated. Although malicious devices may keep the pool drained, an honest client will eventually (probabilistically) succeed in sending an AUTH message to the MDCF through the flood of adversarial messages, thus reserving a provider. Authenticated connections are long-term, so honest devices need only succeed once. When under attack, the time needed for an honest device to connect may be arbitrarily long, but in practice would be bounded with high probability [13].

### 3 Security Design

#### 3.1 Device Authentication Hooks

The MDCF connection state machine, before the introduction of authentication, automatically transitioned from the AUTHENTICATING state to the AUTHENTICATED state (without implementing authentication). To make authentication as seamless as possible for MDCF developers, hooks were placed such that all authentication protocols are executed while in the AUTHENTICATING state. Although authentication protocols themselves may involve multiple rounds of message exchange, this is hidden and encapsulated by the AUTHENTICATING state. The net result is that a full authentication protocol can be implemented without requiring any changes to the remainder of the system, provided any communication channel between the device and Network Controller have already been set up.

To make this possible, hooks are placed so that the protocol is executed after the AUTH message is exchanged but before the AUTHACCEPTED message is sent. These messages are sent and received within the DISCONNECTED and AUTHENTICATING states. Therefore, the primary location for the hooks is within the connection state machine. Our API does not enforce any restrictions

on either the number of rounds or the content of messages exchanged by authentication providers. Thus the framework satisfies the requirement that developers are allowed to implement arbitrary authentication protocols. Because it is possible to implement arbitrary authentication protocols, developers may create providers which facilitate server authentication in addition to the standard client (device) authentication. An authentication provider factory is initialized at runtime, and a provider pool (with at least one provider) is ready when a device calls the provider at connection time. The provider is fetched by name (requested by the connecting device) from the pool and executed from within the DISCONNECTED and AUTHENTICATING states. The AUTH message contains an **authentication** object that stores two pieces of information: a value specifying the authentication protocol requested by the device, and the device's public key.

In addition to the hooks described above, an additional message must be sent from the Network Controller to the device across the atrium channel (used by all devices upon initial connection). This message, called the AUTH\_PROTOCOL message, was required as part of this implementation. It contains information about the channels that will be used by the authentication provider to execute an authentication protocol and may also include other information such as the public key of the Network Controller. The authentication providers have access to two channels with which they can communicate with each other. The NULL authentication providers currently make use of the public atrium channels, `from_mgr_atrium` and `to_mgr_atrium`. It should be noted that these channels were used out of convenience for this initial implementation, but they are ill suited for this purpose because every message sent on `from_mgr_atrium` is effectively broadcast to all other devices leading to unnecessary communication overhead. Future authentication providers will use private channels instead. Authentication providers execute their protocol when their respective `runAuthProtocol()` methods are called. (Note that this is not a security risk, as all messages between the MDCF and the device would be end-to-end encrypted and authenticated in a production system.) The connection state machines wait for these methods to return boolean values to indicate either a successful or failed authentication attempt. If the authentication fails then the device is disconnected, otherwise, the device resumes the connection process.

The location of the authentication hooks also allows us to ensure that authentication can not be bypassed (unless disabled totally by a system administrator). The only way that a device can connect to the MDCF is by progressing through the correct sequence of states in the connection state machine. The sequence of states can be seen in Fig. 3. A device begins in the DISCONNECTED state. From there, it may only transition to the AUTHENTICATING state. A device in the AUTHENTICATING state must enter the DISCONNECTED state if authentication fails, or AUTHENTICATED if it succeeds. Each device must go through the AUTHENTICATING state in order to connect. The authentication hooks are placed such that an authentication provider must execute before a

device can transition from AUTHENTICATING to AUTHENTICATED, therefore authentication may not be bypassed.

### 3.2 Message Confidentiality and Authenticity Hooks

In addition to the authentication hooks, we have added hooks for channel security providers. These providers allow us to gain confidentiality through message encryption, and enforce and verify the integrity and authenticity of messages using digital signatures or message authentication codes. (Although the channel security providers are an essential part of the overall MDCF security architecture, we primarily focus on evaluating authentication providers in this work.)

To gain these additional security properties, we position hooks within the channel service so that every message sent or received by a device or app must pass through a channel security provider. Each channel security provider is able to apply arbitrary transformations on a message, so we can transparently encrypt, decrypt, and authenticate messages. During the initialization of a message sender or receiver, it is bound to a channel security provider, ensuring that security-related transformations on messages may not be bypassed. Channel security providers come in pairs – one for sending a message and another for receiving a message. The application of these providers is on a channel-by-channel basis, making it possible to extend this feature to provide fine-grained control over how confidentiality, integrity, and authenticity are enforced for each channel. We envision that a device’s long term keys, obtained during device authentication, will be stored within the Device Registry. These long term keys might then be used in some way to derive short term keys for encryption and message authenticity/integrity. To date, we have implemented a NULL channel security provider, which performs no operations on a message, and another one which uses Java’s built in TLS provider, which is passed to the channel security providers from the TLS authentication provider.

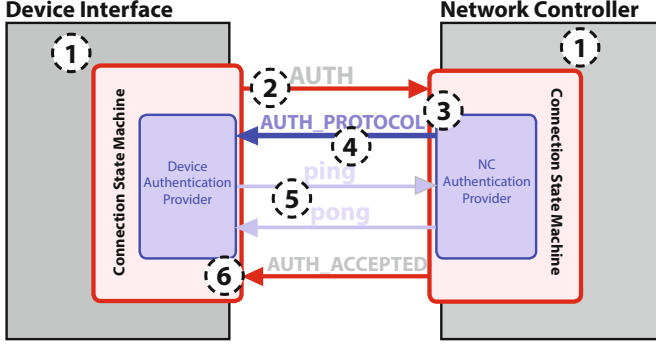
### 3.3 NULL Authentication Provider

To check the overhead of our design and the expressiveness of the hooks, we implemented a NULL authentication provider, which authenticates successfully if it receives a message “PONG” in response to its challenge “PING.” The execution of the provider is mapped out in Fig. 4 and described below.

1. The MDCF and device initialize. The MDCF initializes providers and populates its provider pool.
2. The device fetches any<sup>1</sup> supported authentication method from the device-side authentication provider object, then composes and sends an AUTH message to the Network Controller (NC) with the name of the authentication algorithm.

---

<sup>1</sup> Multiple methods may be supported both by the device and the MDCF, but currently negotiation is not implemented.



**Fig. 4.** Illustration of the authentication process using the NULL provider.

3. Upon reception of the AUTH message, the NC creates its own view of the connection state machine for this device. It then fetches the appropriate pre-initialized authentication provider from the provider pool, and passes the contents of the AUTH message to this provider.
4. The NC-side authentication provider obtains channels to be used for executing the authentication protocol with the device. It then sends an AUTH\_PROTOCOL message to the device that specifies which channels should be used by the authentication protocol only (the device is assigned new channels after it successfully authenticates).
5. The protocol is executed, for the NULL authentication provider, which simply consists of an exchange of the strings “PING” and “PONG” between the device and NC.
6. Upon successful completion of the protocol, the NC creates new private channels for the device, sending the handles of those channels to the device in an AUTH\_ACCEPTED message.

In addition to our NULL provider, we implemented three “non-trivial” providers for evaluation purposes: SSL/TLS, DSA, and DSA+DH. This exercise allowed us to confirm that we meet two of our stated requirements. We found that the API is sufficiently expressive, powerful, and easy to use. Also, as we explain later in this section, we found the only source of overhead comes from the authentication modules themselves. Compared to developing and implementing the framework, creating a provider is relatively simple, e.g. our SSL/TLS provider is only 207 lines of Java code. It is based on Oracle’s `java.net.ssl` implementation, running TLS 1.2 and using the `TLS_DHE_DSS_WITH_AES_128_CBC_SHA` cipher suite. This implementation can be trivially expanded to support mutual authentication with only a few additional lines of code (SSL/TLS provides this as standard functionality). Code length metrics for all providers are in Table 1. The other two authentication providers (DSA and DSA+DH) use a simple challenge-response protocol in which a message from the device to the Network Controller includes a DSA signature from the device. Upon receiving this message, the Network Controller verifies the signature and then sends a signed response message

**Table 1.** Authentication provider (device- and MDCF-side), complexity measured using lines of code (LOC), and complexity increase from the NULL provider. NULL is little more than the common infrastructure/scaffolding. The “Increase over NULL” column is therefore a more accurate representation of the code complexity increase of new authentication modules.

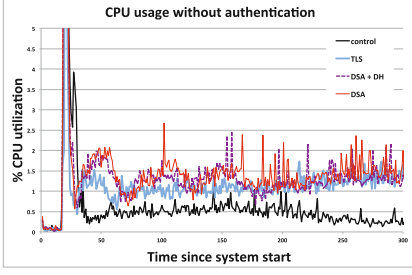
Provider	Implementation (LOC)		Increase over NULL	
	MDCF	Device	MDCF	Device
NULL	128	72	1	1
DSA	151	120	1.18	1.67
DSA+DH	200	178	1.56	2.47
SSL/TLS	207	171	1.62	2.38

to the device. Once the device verifies this response, the authentication protocol terminates – note that this is a mutual authentication protocol.

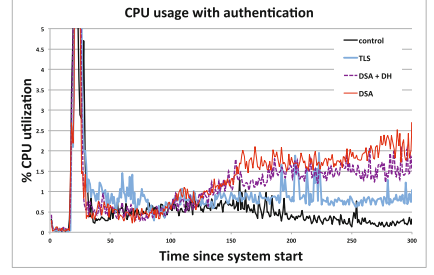
## 4 Evaluation

In order to confirm that we meet the requirement of overhead in the authentication system stemming only from the authentication modules themselves, we ran performance tests of our modified MDCF implementation on a server with dual hex-core Intel Xeon X5670 64-bit CPUs at roughly 2.93 GHz, with 12 MB cache and 24 GB system RAM, running Linux 3.8.13 and Sun’s Java virtual machine version 1.7.0\_21. The resulting performance is shown in Fig. 5. Due to the limitations of Java and the current MDCF architecture on our testbed, we could only reliably test 340 or fewer concurrent devices. In an attempt to tax the resources of the MDCF and our authentication providers, the initial sharp spike in resource usage is due to all test devices attempting to connect simultaneously. Each device begins sending physiological data (SpO2 and pulse rate) following a successful join.

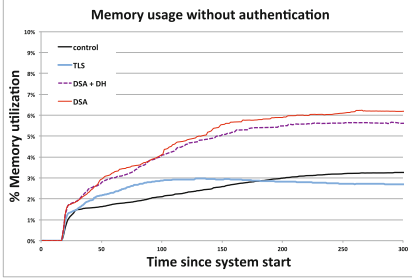
Figures 5(a) and (c) show the resource usage of MDCF using unauthenticated connections. The Y axis are constrained for readability. The highest observed CPU utilization within the startup “spike” was 16% (DSA). Figures 5(b) and (d) show the resources used after including framework hooks only (control) and when using various authentication providers. The highest CPU utilization within the startup spike was 19.75% (TLS), with DSA and DSA+DH reaching 17.4% and 16.7%, respectively. Each line represents an average of 11 instances of tests using identical configurations with 340 devices (device-side performance not shown for readability). The standard error is negligible (the difference between lines is statistically significant), and error bars have been omitted for clarity. The control is a version of MDCF without any authentication code at all – the authentication code was not disabled, but rather removed entirely to avoid unexpected interactions.



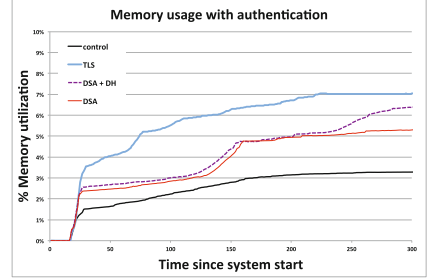
(a) MDCF processor usage with all devices permitted



(b) MDCF processor usage with only authenticated devices permitted



(c) MDCF memory usage with all devices permitted



(d) MDCF memory usage with only authenticated devices permitted

**Fig. 5.** MDCF resource usage with 340 virtual devices running on a different host.

The entire authentication framework consumes negligible resources – indistinguishable from control, satisfying our requirement. Authentication modules in Figs. 5(a) and (c) show a modest but fixed resource cost. They are included in the running code, and a fixed number are initialized to populate the provider pool, but are inactive – devices do not include authentication code and therefore never request to authenticate (backward compatibility mode). Running authentication modules impose an increase in resource usage dependent on the specific protocol being used (resource usage is protocol-dependent).

The network overhead in terms of latency and traffic volume is highly dependent on the individual protocol being used, and can be tuned (by selecting the appropriate protocol) depending on requirements. Authentication imposes a one-time latency increase due to the larger number of network round trips required at connection time, but the observable slowdown is negligible, and only occurs once – upon initial device connection. We found that only the TLS provider caused an increase in bandwidth usage, but to such a small extent as not to interfere with normal operation.

## 5 Conclusion

In this work we extended the existing MDCF high-assurance medical coordination middleware to add a flexible and modular authentication framework, and showed that, in practice, the framework scaffolding itself adds negligible overhead at execution time. The set of hooks introduced into the MDCF, while minimal, is nonetheless sufficiently expressive to support the design and integration of arbitrary modules implementing arbitrary authentication protocols using arbitrarily many rounds of communication before passing control back to the body of the MDCF. Moreover, the code is fully concurrent – devices do not have to wait “in line” to authenticate, but are handled at the same time. Authentication requests can be processed simultaneously, limited only by the performance of the MDCF itself, as authentication scaffolding did not result in significant overhead.

While we do not implement the authentication protocols that we expect to be used in practice (ours are somewhat simplified), we nonetheless observe that the framework itself does not impose undue burden on the coordination middleware, and therefore the performance of future security modules will be bounded by the efficiency of those protocols themselves and their individual implementations, not the cost of dynamic dispatch and call-at-runtime semantics of the modular security framework. Auto-generation of device module code from MDCF-side code, as well as implementing and measuring individual authentication protocol performance on the MDCF and device sides is left up to future work. Further, our current providers rely on pre-shared certificates, but in practice this may not be the case – a “true” Plug-n-Play device, connecting for the first time, will have to transmit its certificate, leading to greater network overhead. Developing and evaluating full certificate trust chain verification is likewise future work.

**Acknowledgments.** The authors would like to thank Daniel Andresen for his input and help in testing the prototype. The computing for this project was performed on the Beocat Research Cluster at Kansas State University, which is funded in part by NSF grants CNS 1006860, EPS 1006860, and EPS 0919443. This research was supported in part by the NIH grant 1U01EB012470-01 and NSF awards CNS 1126709, CNS 1224007, and CNS 1253930.

## References

1. Hatcliff, J., Vasserman, E., Weininger, S., Goldman, J.: An overview of regulatory and trust issues for the integrated clinical environment. In: Joint Workshop On High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS/MD PnP) (2011)
2. Hatcliff, J., King, A., Lee, I., MacDonald, A., Fernando, A., Robkin, M., Vasserman, E.Y., Weininger, S., Goldman, J.M.: Rationale and architecture principles for medical application platforms. In: International Conference on Cyber-Physical Systems (ICCPs) (2012)
3. Conmy, P., Nicholson, M., McDermid, J.: Safety assurance contracts for integrated modular avionics. In: Australian Workshop on Safety Critical Systems and Software (SCS), vol. 33 (2003)

4. ASTM Committee F-29, Anaesthetic and Respiratory Equipment, Subcommittee 21, Devices in the integrated clinical environment: Medical devices and medical systems – essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) (2009)
5. King, A., Procter, S., Andresen, D., Hatcliff, J., Warren, S., Spees, W., Jetley, R., Jones, P., Weininger, S.: An open test bed for medical device integration and coordination. In: International Conference on Software Engineering (ICSE) (2009)
6. Arney, D., Weininger, S., Whitehead, S.F., Goldman, J.M.: Supporting medical device adverse event analysis in an interoperable clinical environment: design of a data logging and playback system. In: International Conference on Biomedical Ontology (ICBO) (2011)
7. Gong, L., Ellison, G.: Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation, 2nd edn. Pearson Education, Upper Saddle River (2003)
8. OpenSSL: OpenSSL: Documents, ssl(3) (2012). <https://www.openssl.org/docs/ssl/ssl.html>
9. McCarty, B.: SELinux: NSA's Open Source Security Enhanced Linux. O'Reilly, Sebastopol (2005)
10. Glenn, R., Kent, S.: The NULL encryption algorithm and its use with IPsec (1998)
11. Schuba, C.L., Krsul, I.V., Kuhn, M.G., Spafford, E.H., Sundaram, A., Zamboni, D.: Analysis of a denial of service attack on TCP. In: IEEE Symposium on Security and Privacy (1997)
12. Snyder, B., Bosanac, D., Davies, R.: ActiveMQ in Action. Manning Publications, Manning Pubs Co Series, Manning (2011)
13. Millen, J.K.: A resource allocation model for denial of service. In: IEEE Symposium on Security and Privacy (1992)

Software Engineering in Health Care  
4th International Symposium, FHIES 2014, and 6th  
International Workshop, SEHC 2014, Washington, DC,  
USA, July 17-18, 2014, Revised Selected Papers  
Huhn, M.; Williams, L. (Eds.)  
2017, X, 239 p. 75 illus., Softcover  
ISBN: 978-3-319-63193-6