Communication Patterns for Interconnecting and Composing Medical Systems (Extended Version)

Venkatesh-Prasad Ranganath Yu Jin Kim John Hatcliff Robby
{rvprasad, yujin, hatcliff, robby}@k-state.edu
CS Department, Kansas State University, U.S.A.

Abstract— **This paper proposes a set of communication patterns to enable the construction of medical systems by composing devices and apps in Integrated Clinical Environments (ICE). These patterns abstract away the details of communication tasks, reduce engineering overhead, and ease compositional reasoning of the system. The proposed patterns have been successfully implemented on top of two distinct platforms (i.e., RTI Connext and Vert.x) to allow for experimentation.**

I. Introduction

The ability to compose various devices and apps (applications) into a medical system at point-of-care is similar to the ability to compose a software system by combining off-the-shelf software components. As with component-based approach to software [16], a component-based approach to medical systems depends on the *description of capabilities and interfaces of devices and apps* and the ability to check for *substitutability* (e.g., can pulse oximeter X be replaced with pulse oximeter Y without perturbing the medical system?) and *compatibility* (e.g., can pulse oximeter X be composed with patient monitor app M?) using device and app descriptions. These requirements are necessary to ensure *flexible and correct composition of interoperable devices and apps.*

Unlike most day-to-day software, failures in medical systems can have adverse consequences including loss of life. Thus, a component-based approach to medical systems should also provide support to reason about *the safety of both the components (i.e., devices and apps) and the composition (i.e., composed medical systems).*

In this regard, the ASTM F2761 standard proposes an architecture for *integrated clinical environments (ICE)* [9] that enable the composition of devices and apps at point-of-care into a medical system. Complementing ASTM F2671, the AAMI-UL JC 2800 effort is developing a family of standards for safety, security, and essential performance of interoperable medical systems [4]. The 2800 standards include safety/security requirements for the ICE architecture, as well as safety/security process-related requirements for the development of interoperable systems built from ICE compliant devices and apps. While these standards recognize the need of device models — description of the capabilities and interfaces of devices[1], there is very little information about what should constitute a device model. In a recent effort, driven by the perceived needs of the stakeholders in ICE and the goals laid out for medical application platforms (MAP) [10] such as ICE (e.g., device/app-wise regulation, dynamic composition of devices and apps), we identified a set of requirements for device models [13].

The *description of communication needs of a device* was identified as one of the requirements of the device model. Consequently, we proposed a set of communication patterns that can serve as the schema to describe the communication needs of devices/apps. In this paper, we present these communication patterns.

[1]These models are applicable to apps as well.

II. Rationale for Communication Patterns

Before we describe communication patterns, we describe the rationale for including and excluding various aspects of communication from the patterns.

Data Type: For a valid connection between two communication endpoints, both need to agree on the types of the data being communicated. In programming languages community, there are numerous efforts pertaining to the design and use of type systems to reason about data types. Since this knowledge can be reused, the proposed patterns do not prescribe how to capture or reason about data types.

Quality of Service (QoS): Communications in medical systems may require and impose strict guarantees, e.g., the x-ray machine should be stopped after 2 seconds. Such constraints can impact the behavior of the underlying communication substrate and the communicating components. Furthermore, violation of such constraints can lead to catastrophic results, e.g., the stop command may not reach the x-ray machine in time and cause harmful prolonged exposure. Hence, the proposed patterns capture QoS requirements.

Local Control: Not all QoS properties are supported by all communication substrates. To deal with this possibility, the proposed patterns breakdown common QoS properties into finer properties that can be monitored locally (as part of the client or the service) and from which common QoS requirements for the underlying communication substrate can be derived.

Abstraction: In component-based approach to software, component frameworks abstract away lower-level details of various aspects, e.g., communication, data persistence. Such abstraction helps component developers to focus on the core behavior of components and delegate lower level details to the framework. Moreover, such abstraction can assist with modeling and reasoning of the components and their composites. In a similar spirit, the proposed patterns abstract the lower-level details of communication substrate (and are agnostic to communication substrates).

Command-Query Separation: Bertrand Meyer devised *command-query separation (CQS)* principle in the context of Eiffel programming language [15]. This principle states that every method (function) should either be a command that performs an action or a query that returns data, but not both. Such separation aids reasoning about composite systems by providing clear separation between interactions in such systems. Hence, the proposed patterns follow the CQS principle.

III. Communication Patterns

To describe communication patterns, we use the following terms and assumptions.

1) Both devices and apps are referred as *components*.
2) Components communicate with each other via a *communication substrate* (e.g., O/S-level networking level, a sophisticated middleware) that is responsible for moving the bits from source to destination.
3) If a component can reach the communication substrate, then the communication substrate can reach the component and vice versa.
4) Components have two sorts of capabilities: *provide and receive data* and to *perform actions to change the physical state of the system*, e.g., by moving a stepper motor or updating a display.
5) Parameters can influence actions. To avoid the influence of change of parameters when a controlled action is in progress, actions depend on the value of parameters at the time of initiation.
6) Components expose *data interfaces* to provide and access data (including parameters) and *action interfaces* to initiate offered actions.
7) Each component (and, consequently, the composite system) has a *soft (data) state* composed of observable data (i.e., data accessible via data interfaces) and *hard (physical) state* composed of physical state of the component (e.g., the current xy-angle of a robotic arm).
8) Hard states can be modified only via action interfaces while soft states can be directly modified via data interfaces and indirectly modified via action interfaces, e.g., performing an action can change the reported data.
9) Each pattern comprises various roles fulfilled by components. Each component can take on multiple roles.
10) Patterns involving data exchanges rely on a sound type system to ensure the validity of data exchanges.

We describe each pattern in a fixed format that captures the *roles* involved in the pattern, *intent* of the pattern, *description* of the pattern, prescribed *use* of the pattern, *QoS properties* supported by the pattern, *QoS requirements* on the communication substrate for supporting the pattern, *possible failures* in the pattern, an *example* use of the pattern, and any *challenges* in realizing the pattern.

A. Publisher-Subscriber (Producer-Consumer) Pattern

Roles: Publishers (Producers) and Subscribers (Consumers).

Intent: Decouple publishers (producers) and subscribers (consumers) of data by focusing on the topic of interest (and not on the publishers and subscribers).

Description: In this pattern, *publisher* role publishes data about a *topic* and a *subscriber* role subscribes to data about a topic. (This pattern is an incarnation of topic-based communication offered by most publish-subscribe middleware [3].) The topic uniquely identifies the type of the published/subscribed data. The act of publishing data is asynchronous — the publisher does not wait for the communication substrate to deliver the message to subscribers. Further, the subscriber receives messages from every publisher in the order the messages are published by the publisher (local order); out of order messages are dropped.

Use: Connect data interfaces not associated with parameters (that affect actions). This restriction prevents ad-hoc changes to parameters and initiation of actions.

QoS Properties: Figure 1 illustrates the relationship between the supported QoS properties as the pattern is exercised at runtime.

- *MinimumSeparation (N_{pub})* between two consecutive publications. If the duration between two consecutive publications is less than N_{pub}, then the second publication is dropped with *fast publication* failure.[2]
  This property is associated with the publisher.
  While deploying the publisher, N_{pub} is used (along with size of the publication payload) to check the underlying communication substrate can support the network traffic contributed by the publisher.
  At runtime, the communication substrate monitors the interval between consecutive of publication requests, drops the second publication if it deems the publication is faster than specified, and informs the publisher of *fast publication*. The publisher handles fast publication notifications.
- *MaximumLatency (L_{pub})* to accept a publish request. If the communication substrate fails to accept a publish request within L_{pub} time units, then the publication results in *timeout* failure.
  This property is associated with the publisher.
  While deploying the publisher, L_{pub} is used to check if the underlying communication substrate can satisfy the required latency.
  At runtime, the communication substrate accepts the publish request within L_{pub} time units or informs the publisher of *timeout*. The publisher handles timeout notifications.
- *MinimumRemainingLifetime (R_{pub})* of the data upon publication. If the data arrive at the subscriber after R_{pub} time units since publication, then the data is treated as stale and the subscriber is notified of *stale data*.

[2]This QoS property was proposed by King et al. [14].

This property is associated with the publisher.
While deploying the publisher, R_{pub} is used to check if the latency of the underlying communication substrate is low enough to deliver the published data while it is still fresh.
At runtime, the communication substrate tracks the time to transmit the data from the publisher to the subscriber and informs the subscriber of *stale data* if the transmission time exceeds R_{pub}. The subscriber handles stale data notifications.

- *MinimumSeparation (N_{sub})* between two consecutive message arrivals at the subscriber. If the duration between the arrival of two consecutive messages is less than N_{sub}, then the second message is dropped.
This property is associated with the subscriber.
At runtime, the communication substrate monitors the interval between consecutive message arrivals and drops the second message if it deems the message is arriving faster than subscriber expects.
- *MaximumSeparation (X_{sub})* between two consecutive message arrivals at the subscriber. If the duration between the arrival of two consecutive messages is greater than X_{sub}, then the subscriber is notified of *slow publication*.[2]
This property is associated with the subscriber.
At runtime, the communication substrate monitors the interval between consecutive message arrivals and notifies the subscriber of *slow publication* if it deems the messages are arriving slower than subscriber expects.
- *MaximumLatency (L_{sub})* to consume a message. If the subscriber fails to consume a message within L_{sub} time units, then the message is considered as an unconsumed message. After a fixed number of consecutive unconsumed messages (specified by *ConsumptionTolerance* sub-property), the subscriber is notified of *slow consumption*.
This property is associated with the subscriber.
At runtime, the communication substrate monitors the time taken by the subscriber to consume a message and notifies the subscriber of *slow consumption* when this time exceeds L_{sub} for a fixed number of consecutive unconsumed messages. The subscriber handles slow consumption notifications.
- *MinimumRemainingLifetime (R_{sub})* of the received data. If the data arrives at the subscriber with remaining lifetime smaller than R_{sub}, then the data is considered stale and the subscriber is notified of *stale data*.
This property is associated with the subscriber.
While deploying the subscriber, R_{pub}, R_{sub}, and L_m (see *Qos Requirements* below) are used to check if the latency of the underlying communication substrate is low enough to deliver the published data with sufficient remaining lifetime (R_{sub}).
At runtime, the communication substrate tracks the time to transmit the data from the publisher to the subscriber and informs the subscriber of *stale data* if the remaining lifetime of the data is smaller than R_{sub}. The subscriber handles stale data notifications.

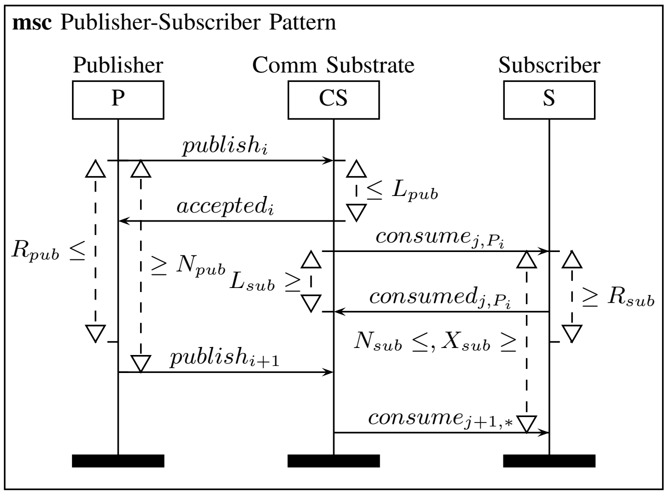

Fig. 1. Sequence of message flows and corresponding time periods (time progressing from top to bottom) in publisher-subscriber pattern.

QoS Requirements: For the data to be delivered with lifetime of at least R_{sub}, communication substrate should ensure maximum message delivery latency (L_m) does not exceed $R_{pub} - R_{sub} - L_{pub} \geq L_m$.

Example: A pulse oximeter broadcasts pulse rate data as a *pulse-rate* topic and a patient monitor app subscribes to this topic to get the patient's pulse rate. Assuming the *pulse-rate* topic is well-defined (based on standards such as 11073 [12]) in terms of the data type and its semantic interpretation, upon switching pulse oximeters, the patient monitor need not be reconfigured if the new pulse oximeter broadcasts its measurement as *pulse-rate* topic.

B. Requester-Responder

Intent: Retrieve data from a specific component.

Description: In this pattern, the *requester* role requests some data from a specific *responder* role and the responder responds back with the data. In terms of data flow, the data flows from the service (responder) to the client (requester). This pattern requires the requester to know the identity of the responder. This identity uniquely identifies the requested data and its type, and this enables static validation of the communication. This pattern is synchronous — the requester waits for either the arrival of the response, a notification of failure, or a fixed period, whichever is earlier. Furthermore, the responder receives requests in the order they are issued by the requester (local order); out of order requests are dropped.

Use: Connect data interfaces including those associated with parameters.

QoS Properties: Figure 2 illustrates the relationship between the supported QoS properties as the pattern is exercised at runtime.

- *MinimumSeparation (N_{req})* between consecutive requests. If the duration between two consecutive requests is less than N_{req}, then the second request is dropped with *fast request* failure.

This property is associated with the requester.
While deploying the requester, N_{req} is used (along with size of the requested payload) to check the underlying communication substrate can support the network traffic contributed by the requester.
At runtime, the communication substrate monitors the interval between consecutive of requests, drops the second request if it deems the publication is faster than specified, and informs the requester of *fast request*s. The requester handles fast publication notifications.

- *MaximumLatency (L_{req})* between the sending of a request and the arrival of the corresponding response. If the response does not arrive within L_{req} time units, then the request results in *timeout* failure.
This property is associated with the requester.
While deploying the requester, L_{req} is used to check if the underlying communication substrate can satisfy the required latency. When configured with a responder, L_{req} is used along with L_{res} to check if the communication substrate and the responder can together satisfy the required latency.
At runtime, the communication substrate accepts the request, delivers the request to the responder, and delivers the corresponding response to the requester within L_{pub} time units or informs the requester of *timeout*. The requester handles timeout notifications.
- *MinimumRemainingLifetime (R_{req})* of the requested data. If response arrives at the requester with remaining lifetime less than R_{req}, then the requester is notified of *stale data*.
This property is associated with the requester.
While configuring the requester with a responder, R_{req}, R_{res}, and L_m (see *Qos Requirements* below) are used to check if the latency of the underlying communication substrate is low enough to deliver the response to the requester with sufficient remaining lifetime (R_{req}).
At runtime, the communication substrate tracks the time to transmit the response from the responder to the requester and informs the requester of *stale data* if the remaining lifetime of the response is smaller than R_{req}. The requester handles stale data notifications.
- *MinimumSeparation (N_{res})* between the arrival of consecutive requests. If the duration between the arrival of two consecutive requests is less than N_{res}, then the request is dropped with *excess load* failure.
This property is associated with the responder.
At runtime, the communication substrate monitors the interval between consecutive request arrivals and drops the second request if it deems the request is arriving faster than responder expects.
- *MaximumLatency (L_{res})* between receiving a request and providing the response to the communication substrate. If the response is not provided within the L_{res} time units, the request results in *timeout* failure.
This property is associated with the responder.
At runtime, the communication substrate monitors the time taken by the responder to consume a request. If this time exceeds L_{sub}, the communication substrate notifies the requester of *timeout*. The requester handles timeout notifications.
- *MinimumRemainingLifetime (R_{res})* of retrieved data (response). If the data with the promised minimum remaining lifetime cannot be provided by the responder, then request results in *data unavailable* failure.
This property is associated with the responder.
While deploying the responder, R_{res} is used to check if the latency of the underlying communication substrate is low enough to deliver the response while it is still fresh.
At runtime, if the responder cannot provide a response with promised minimum remaining lifetime cannot be provided, it informs the communication substrate of *data unavailable*. The communication substrate communicates this information to the requester for proper handling.

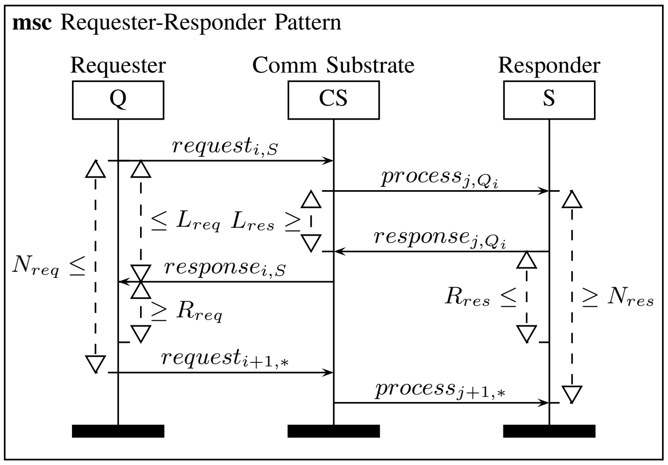

Fig. 2. Sequence of message flows and corresponding time periods (time progressing from top to bottom) in requester-responder pattern.

QoS Requirements: For response to be delivered with lifetime of at least R_{req}, communication substrate should ensure the sum of maximum latencies to deliver the request to the responder (L_m) and the resulting response to the requester (L'_m) does not exceed $L_{req}+R_{req}-L_{res}-R_{req} \geq L_m+L'_m$.

Possible Failures: In addition to QoS property specific failures, the pattern can fail with *data unavailable* failure when the responder cannot provide the requested data (due to reasons such as the data are being calculated or the state of the responder inhibits the provision of the data).

Example: A blood pressure (BP) monitor periodically measures blood pressure and responds with the current measurement when requested via its uniquely identified data interface. A patient monitor app requests blood pressure measurement from the BP monitor via its uniquely identified data interface. Upon switching the BP monitor, the patient monitor will have to be reconfigured with the unique identifier of the data interface of the new BP monitor.

C. Sender-Receiver

Intent: Provide data to a specific component.

Description: In this pattern, the *sender* role sends data to a specific *receiver* role and the receiver responds back with either *data accepted* or *data rejected* acknowledgement. In terms of data flow, the data travels from the client (sender) to the server (receiver). This pattern requires the sender to know the identity of the receiver. This identity uniquely identifies the sent data and the data type; this enables static validation of the communication. This pattern is synchronous — the sender waits either for an acknowledgement, a notification of failure, or a fixed period, whichever is earlier. Furthermore, the receiver receives data in the order they are sent by the sender (local order); out of order requests are dropped.

Use: Connect data interfaces associated with parameters.

QoS Properties: Since this pattern facilitate point-to-point communication, it supports QoS properties similar to those supported by the requester-responder pattern. However, being limited to parameters, it supports fewer QoS properties. Specifically, the sender role supports *MinimumSeparation (N_{sen})* and *MaximumLatency (L_{sen})* properties similar to *MinimumSeparation (N_{req})* and *MaximumLatency (L_{req})* properties supported by the requester role with *fast send* and *timeout* failures, respectively. Similarly, the receiver role supports *MinimumSeparation (N_{rec})* and *MaximumLatency (L_{rec})* properties similar to *MinimumSeparation (N_{res})* and *MaximumLatency (L_{res})* properties supported by the responder role with the same kinds of failures.

QoS Requirements: To support the above QoS properties, the communication substrate should ensure the sum of maximum latencies to deliver the sent data to the receiver (L_m) and the resulting acknowledgement to the sender (L'_m) does not exceed $L_{sen} - L_{rec} \geq L_m + L'_m$.

Example: A BP monitor measures a patient's blood pressure at regular interval of 3 minutes. Upon attaching an infusion pump to the patient, the patient monitor app instructs the BP monitor to change the measurement interval to 1 minute by sending the new interval parameter.

D. Initiator-Executor

Intent: Initiate an action in a specific component.

Description: In this pattern, the *initiator* role requests a specific *executor* role to perform an action. Depending on the successful completion of the action, the executor provides *action succeeded* or *action failed* acknowledgement. If the action is unavailable, then the executor provides *action unavailable* acknowledgement. This pattern requires the initiator to know the identity of the executor. Since the pattern does not facilitate flow of parameters, it is safe to combine this identity with an action identifier provided by the initiator to uniquely identify the action. This pattern is synchronous — the initiator waits either for an acknowledgement, a notification of failure, or a fixed period, whichever is earlier. Also, the executor receives initiations in the order they are issued by the initiator; out of order requests are dropped.

Use: Connect action interfaces.

QoS Properties: This pattern supports QoS properties similar to those supported by the sender-receiver pattern. Specifically, the initiator role supports *MinimumSeparation (N_{ini})* and *MaximumLatency (L_{ini})* properties similar to *MinimumSeparation (N_{sen})* and *MaximumLatency (L_{sen})* properties supported by sender role but with *fast initiation* and *timeout* failures, respectively. Similarly, the executor role supports *MinimumSeparation (N_{exe})* and *MaximumLatency (L_{exe})* properties similar to *MinimumSeparation (N_{rec})* and *MaximumLatency (L_{rec})* properties supported by receiver role with the same kinds of failures.

QoS Requirements: To support the above QoS properties, the communication substrate should ensure the sum of maximum latencies to deliver the initiation to the executor (L_m) and the resulting acknowledgement to the initiator (L'_m) does not exceed $L_{ini} - L_{exe} \geq L_m + L'_m$.

Example: Before capturing a radiograph of a patient hooked to a ventilator, a coordination app initiates a stop action on the ventilator attached to the patient being radiographed.

E. Orchestration

Intent: Orchestrate data transfers and actions among devices and apps to accomplish a task (or achieve a goal).

Description: In this pattern, a component playing the role of the *orchestrator* interacts with various components via different communication patterns in the specified order. So, unlike previous patterns, the number of participants can vary in this pattern. The success and failure of orchestration is dictated by the success and failure of communication patterns constituting the orchestration. The same is true of the QoS requirements imposed by the orchestration on the communication substrate. As for QoS properties, *MaximumLatency* to complete an orchestration is the only relevant property and it too depends on constituting communication patterns.

Example: To capture a radiograph of a patient hooked to a ventilator, once the patient is in place, the orchestrating app stops the ventilator, starts the x-ray machine, stops the x-ray machine, and then restarts the ventilator. If the x-ray machine fails to start, the app restarts the ventilator.

Challenges: Unlike other patterns that have specific failures, each orchestration can have different failures stemming from constituting communication patterns. Further, when an orchestration fails after executing few constituting patterns, it is possible the system can be in an unsafe state, e.g., x-ray machine may fail to execute the *stop* action. Worse yet, the system could have performed some action that cannot be undone, e.g., performed a drug injection amongst a series of drug injections and a constituting communication pattern fails. Such concerns are further exacerbated when orchestrations performed in parallel interfere with each other as they involve common components.

Above situations are strikingly similar to situations involving concurrent modifications in databases. The ill-effects

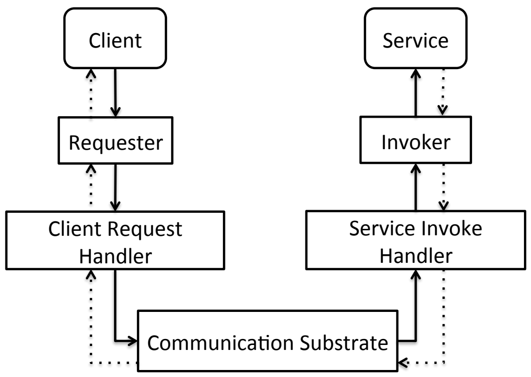

Fig. 3. Architecture of our implementation of communication patterns. Solid lines represent flow of data/request from client to service. Dotted lines represent the flow of data/response from the service to client.

of concurrent modifications (data inconsistency) are avoided in databases by employing transactions, which provide isolation and atomicity (all-or-nothing) guarantees by relying on the ability to rollback (undo modifications). So, while transactions seem to be an obvious solution to safely realize orchestration in medical systems, the physical aspect of medical systems limits the ability to undo modifications and, consequently, the possibility of directly applying the concept of transactions to medical systems.

To move forward, we need a better understanding of the sort of orchestration needed in medical systems. Specifically, we need to understand clinical scenarios that need orchestration, characteristics of such scenarios, and requirements of orchestration in such scenarios in terms of the sort of orchestrated actions, their effects, possibilities to undo effects, and admissible failures and their characteristics (i.e., nature of failure, criticality). With this understanding, we can devise and evaluate alternative approaches to support the desired sorts of orchestrations in a given clinical scenario.

IV. Implementation

To validate the viability of the proposed patterns, we implemented the patterns on two different platforms in Java (RTI Connext [7] and Vert.x [8]). The source code of the implementation is available at http://bitbucket.org/rvprasad/clinical-scenarios/.

A. Architecture

As the entry point, the implementation exposes a `CommunicationManager` interface to configure (e.g., via `InitiatorConfiguration`), instantiate, and manage instances of various roles described in the patterns. Once a specific role has been instantiated, actions supported by that role instance can be accessed via a simple interface, e.g., `Initiator` (as shown below).

```
interface Initiator<T extends Serializable> {
  InitiationStatus initiate(T action);
}
```

Under the hood, we use a layered architecture based on *remoting patterns* [17] to implement the patterns. As depicted in Figure 3, the implementation uses the *requester* remoting pattern to encapsulate and hide details specific to client-side processing pertaining to communication, e.g., check for maximum service latency. Similarly, the implementation uses *invoker* remoting pattern to encapsulate and hide details specific to service-side processing pertaining to communication, e.g., to check for minimum separation between consecutive service requests.

To interface with communication substrates that move bits, our implementation uses *client request handler* and *service request handler* remoting patterns on client and service sides, respectively. These remoting patterns allow the implementation to decouple the details of the proposed patterns from the low-level details of various substrates. Consequently, to support the proposed patterns on top of a different communication substrate, we only need to implement these handlers on top of the substrate along with a substrate-specific implementation of `CommunicationManager`.

B. Realization

We have successfully tested this architecture by implementing a library of these patterns (except orchestration) on top of RTI Connext and Vert.x. *RTI Connext* is an implementation of *Data Distribution Service (DDS)* [6] that supports publish-subscribe paradigm. In addition, it also supports point-to-point communication via request-reply pattern. So, with RTI Connext, we realized publisher-subscriber pattern directly via existing support for publish-subscribe pattern and the rest of the patterns using request-reply pattern. In comparison, *Vert.x* is a lightweight, high performance application platform for JVM that provides an event bus with support for point-to-point communication. Consequently, with Vert.x, we realized every pattern using the event bus.

Besides decoupling the implementation from communication substrate and simplifying the application-level communication interface, the architecture allows the implementation to non-intrusively add logic to collect data about communication. For example, the architecture allows seamless logging of data about communication at each role and client/service that can help with analysis (e.g., diagnostic, forensic) of medical systems in various clinical scenarios.

C. Supporting QoS Properties

To understand how these patterns and their realizations can support QoS properties, consider an alternate view of the layered architecture of our implementation of publish-subscribe pattern as shown in Figure 4.

In this architectural view, we assume the implementation uses a middleware. So, each publishing (subscribing) component is associated with a middleware entity/layer that acts as the bridge between the network and the component. The middleware and the network together are considered as the communication substrate while the other layers are consdiered as part of the communicating components.

To publish data, the publishing component uses the *SCP publisher API*. This triggers the *SCP Service Layer* (denoted by *b*) to check various QoS properties (e.g., the duration between previous *a'* and *a* exceeds N_{pub}) and flags any

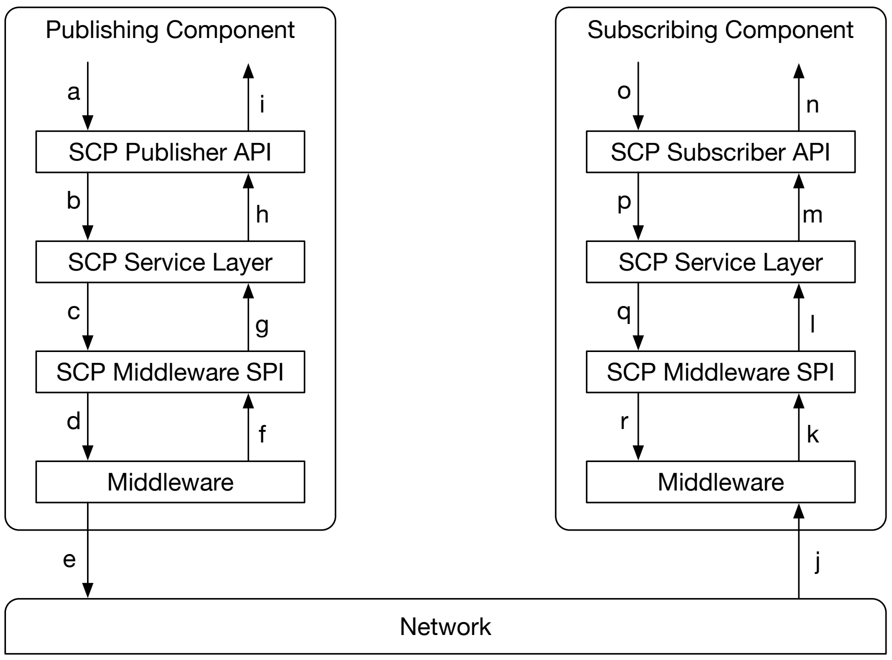

Fig. 4. Alternative view of the architecture of our implementation of Publish-Subscribe communication patterns. Arrows represent the flow of control.

violation as runtime failures (e.g., *fast publication* along the control flow path *a-b-h-i*.) If every check is successful, then *SCP Service Layer* uses *SCP Middleware SPI* (denoted by *c*) to request the *Middleware* (denoted by *d*) to perform the actual publication (denoted by *e*). In addition, *SCP Service Layer* monitors the time taken by *SCP Middleware SPI* to complete the publication request. If this duration does not exceed L_{pub}, then the publication request completes successfully along the control flow path would be *a-b-c-d-(f-g-h-i|e)* where *f-g-h-i* occurs concurrently with *e*. If the duration exceeds L_{pub}, then *timeout* failure is flagged to the publishing component along the control flow path would be *a-b-h-i*.

On the subscribing end, upon receiving published data, *middleware* uses *SCP Middleware SPI* (denoted by *k*) to trigger *SCP Service Layer* (denoted by *l*) to handle the data. *SCP Service Layer* checks various QoS properties (e.g., the duration between previous *l'* and *l* exceeds N_{sub}) and appropriately handles violating situations (e.g., drops the data along the control flow path *j-k-l-q-r* or flags *slow publication* along the control flow path *j-k-l-(q-r|m-n-o-p)*. If every check is successful, then the subscribing handler is invoked (denoted by *n*) via *SCP Subscriber API* (denoted by *m*) along the control flow path *j-k-l-(q-r|m-n-o-p)*.

Unlike in *pub-sub* pattern, the control flow in *req-res*, *sen-rec*, and *ini-exe* patterns can be more involved as they are synchronous and some of the control flow paths can span both ends of the communication. Even so, the checks to support QoS properties are still performed locally in the *SCP Service Layer*.

V. Influence on Process

To understand how these patterns influence the steps of processes involving communicating components, consider the following simple scenario.

- Vendor P develops a device M_p that publishes data of type T.
- Vendor S develops a device M_s that subscribes to data of type T.
- Organization H has a procedure in which devices M_p and M_s are used and configured to communicate with each other.
- P, S, and H use a common middleware W.

Development: While developing device M_p, vendor P should fix N_{pub}, L_{pub}, and R_{pub} QoS properties. Since vendor P has complete control over both hardware and software of M_p, vendor P should ensure by construction and/or testing that the device does indeed satisfy these QoS properties in conjunction with middleware W. Further, external certification agencies could be used to verify and certify the device does indeed satisfy the properties. A similar approach can be adopted by vendor S when developing device M_s.[3]

Procurement: Organization H does not have control of any of the QoS properties of M_p or M_s. However, it should still ensure that devices used in it are compatible. To achieve this goal, while procuring devices, organization H can use the QoS properties of devices to ensure they can be compatible when used together; of course, we assume the organization only procures devices compatible with the middleware used in it.

[3]To reduce the burden of verification, it is possible to envision the service and middleware layers can be verified along with the corresponding middleware independently before they are used by device vendors. However, such verification will be limited to behavioral properties that are not dependent on the capacity of the underlying hardware, e.g., speed. Since most of the QoS properties are time related and tied to hardware capacity, we will have to verify the devices satisfy the properties.

Deployment: Even when devices are compatible, they can fail to operate as desired. Specifically, organization H controls the network thru which devices communicate and the corresponding network latency L_m interacts with QoS properties R_{pub}, R_{sub}, and L_{pub} of the deployed devices. So, when deploying devices M_p and M_s, organization H should ensure network latency L_m will not exceed $R_{pub} - R_{sub} - L_{sub} \geq L_m$. Depending on the kind of deployed network, this could be done by dynamic network configuration or capacity planning.

Other patterns exert similar influence on various steps of processes involving communicating components. Unlike pub-sub pattern, other patterns are synchronous. Hence, their influence on the process may be relatively more in terms of constraints imposed on various activities and roles, i.e., vendors and organizations.

VI. Related Work

In his master's thesis, Hoffman proposed a standard for interoperability of devices and apps in an integrated clinical environment. This standard supported four types of message transactions/exchanges between medical devices — *get* and *set* data, initiate *action*, and notify about *events* [11]. In comparison, the proposed patterns are similar to get, set, and action exchanges. While event exchange is not supported by a dedicated pattern, it can be trivially realized via publisher-subscriber pattern or sender-receiver pattern. Also, the proposed patterns support communication QoS properties and failures relevant in medical systems.

In the ICE community, OpenICE [1] uses publish-subscribe paradigm [3] to facilitate all communications between medical devices and apps in an ICE system. On the other hand, OpenSDC [2] embraces service-oriented architecture via web services and proposes a small set of standard services (e.g., *get, set/action, event report, waveform, PHI (protected health information)*) along with a predefined set of operations. Furthermore, neither of these efforts capture QoS requirements pertaining to communication. In comparison to OpenICE and OpenSDC, we propose a collection of patterns with specific and distinct intent to enable unambiguous description of communication between devices and apps, including QoS requirements. With such specificity and clarity in description, we believe reasoning about medical systems will be easier; specifically, as the community embraces a model-centric process to develop, certify, approve, and deploy safe medical devices and apps. Furthermore, as these patterns abstract away low-level networking details, stakeholders dealing with device/app code can operate at the same level of abstraction as stakeholders dealing with device/app models.

Constrained Application Protocol (CoAP) [5] was recently proposed as the communication protocol in the context of *Internet of Things (IoT)*. The proposed patterns are similar to CoAP — they address the need to describe communication between entities. However, besides the stark difference of the patterns being informative and CoAP being normative, there are few other non-trivial differences. For example, the proposed patterns specify required features of the underlying communication substrate while CoAP prescribes specific transport protocols. Also, unlike CoAP, the patterns directly embody QoS properties and failures relevant to the safe operation of medical systems.

VII. Ongoing Work

Currently, we are implementing few devices and apps using the above described implementation of the proposed patterns to realize various clinical scenarios in an artificial setting. In this exercise, we plan to evaluate the ability of the proposed patterns to facilitate easy logging and diagnostics of medical systems composed of apps and devices; specifically, during forensic analysis of failed clinical scenarios.

In parallel, we are realizing the same clinical scenarios using traditional publish-subscribe pattern using OpenICE [1] and RTI Connext. The goal of this exercise is to compare patterns-based communication to traditional publish-subscribe paradigm both in terms of ease of development and forensic analysis of clinical scenarios.

VIII. Future Work

As mentioned in Section III, orchestration is relevant for effective use of medical systems in many clinical scenarios. However, the physical aspect of medical systems complicates the support for orchestration due to the inability to undo certain actions. So, an open problem in this space is to *identify the kind of orchestration needed in medical systems and then devise and evaluate alternative approaches to support such orchestration.*

IX. Summary

In this paper, we have presented reasons for identifying communication patterns necessary to compose safe and reliable medical systems from devices and apps. As a solution, we have proposed a set of communication patterns (backed by a proof of concept implementation) that can facilitate reliable composition of medical systems. While we proposed orchestration as a communication pattern, we acknowledge more work is needed to understand and support orchestration in medical systems.

With this effort, we see an opportunity in the community to compile and standardize a core set of communication patterns that enable construction of safe and reliable medical systems at point-of-care by composing heterogeneous yet interoperable devices and apps.

Acknowledgement

We thank Stephen Barrett, Andrew King, Insup Lee, Sam Procter, and Eugene Vasserman for their feedback during the discussions of communication patterns.

References

[1] OpenICE Prototype: A New, Open, Interoperable Medical Device Clinical Research Platform. Software available at http://mdpnp.org/MD_PnP_Program___OpenICE.html.

[2] OpenSDC Communication Library. Software available at http://opensdc.sourceforge.net/.

[3] Publish–Subscribe Pattern. Info at http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

[4] Scope of UL2800. Info at http://ulstandardsinfonet.ul.com/stp/\Proposed_Scopes/prop-2800_scope.html.

[5] Constrained Application Protocol (CoAP), June 2014. Info at https://datatracker.ietf.org/doc/rfc7252/.

[6] Data Distribution Service (DDS) specification, April 2015. Available at http://www.omg.org/spec/DDS/1.4/.

[7] RTI Connext, 2015. Available at http://www.rti.com/products/dds/index.html.

[8] Vert.x, 2015. Available at http://vertx.io/.

[9] ASTM International. ASTM F2761 - Medical Devices and Medical Systems - Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE), 2009.

[10] J. Hatcliff, A. King, I. Lee, A. MacDonald, A. Fernando, M. Robkin, E. Vasserman, S. Weininger, and J. M. Goldman. Rationale and Architecture Principles for Medical Application Platforms. In *International Conference on Cyber-Physical Systems (ICCPS)*, pages 3–12, 2012.

[11] R. M. Hofmann. Modeling medical devices for plug-and-play interoperability. Master's thesis, MIT, June 2007.

[12] ISO/IEEE. *ISO/IEEE11073-x Medical Health Device Communication Standards Family*. 2004.

[13] Y. J. Kim, J. Hatcliff, V. P. Ranganath, Robby, and S. Weininger. "Integrated Clinical Environment Device Model:Stakeholders and High Level Requirements". In *Medical Cyber Physical Systems Workshop (MCPS)*, 2015.

[14] A. King, S. Chen, and I. Lee. The middleware assurance substrate: Enabling strong real-time guarantees in open systems with openflow. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 133–140, 2014.

[15] B. Meyer. Command Query Separation. Info at http://en.wikipedia.org/wiki/Command%E2%80%93query_separation.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[17] M. Völter, M. Kircher, and U. Zdun. *Remoting Patterns*. John Wiley & Sons, Ltd, 2004.