OverStar: An Open Approach to End-to-End Middleware Services in Systems of Systems

Paul Grace, Yérom-David Bromberg, Laurent Réveillère, Gordon Blair

OverStar: an open approach to end-to-end middleware services in systems of systems

Paul Grace[1], Yérom-David Bromberg[2], Laurent Réveillère[2], and Gordon Blair[1]

[1] School of Computing and Communications, Lancaster University, UK
p.grace@lancaster.ac.uk, gordon@comp.lancs.ac.uk
[2] LaBRI,University of Bordeaux, France
david.bromberg@labri.fr, laurent.reveillere@labri.fr

Abstract. The increasing complexity of distributed systems, where heterogeneous systems are composed to form systems of systems, pose new development challenges. How can core middleware services, e.g. event communication, resource discovery, etc. be deployed and optimised in an end-to-end manner? Further, how can important properties such as interoperability be managed? In this paper we propose OverStar a framework that generates overlay network based solutions from high-level specifications in order to answer these questions. A middleware service is specified as a self-managing overlay network across heterogeneous systems; timed automata specify how the topology of the network is constructed and the data is exchanged. The key contribution is the open access to individual overlay nodes in order to specify additional flow logic, e.g. the translation of messages to support end-to-end interoperability or the filtering of heterogeneous messages to optimise event dissemination. We evaluate OverStar using service discovery and event communication case studies; these demonstrate the ability to compose heterogeneous systems, achieve end-to-end interoperability and simplify the developer's task. Further, a performance evaluation highlights optimisations that can be achieved.

1 Introduction

Overlay networks are increasingly important in underpinning key middleware functions (e,g. service discovery, multicast, and P2P in various disguises). Indeed they are becoming a pervasive feature of middleware technologies, and their management and co-ordination will be a key requirement in future complex systems. Many different types of overlay networks have been developed to provide virtualised network services for particular environments and requirements, e.g, large-scale resource discovery [22] or multicast [23] in high-churn networks. In addition, software frameworks for overlays: P2 [16], Macedon [21], and OpenOverlays [9] provide tools to rapidly create a tailored overlay network, or incorporate an overlay network as an explicit architectural element of middleware. This work is promising but it falls down in underpinning middleware functions in complex distributed systems-of-systems where there are high levels of *heterogeneity* and *dynamic behaviour*, especially in terms of the middleware

protocols used by end systems that need to be composed dynamically. These end systems may utilise heterogeneous middleware services, i.e., different event communication middleware (e.g. STOMP [3] or OpenWire [4]) and different resource discovery protocols (e.g. SLP or Bonjour). Hence, there is a need to manage this heterogeneity, especially with respect to interoperability and optimisation.:

- *End-to-end interoperability.* Heterogeneous local middleware services must interoperate when composed together in order to realise the global functionality of a middleware service.
- *End system optimisations.* It should be possible to apply service optimisations at the end systems despite the heterogeneous technologies, e.g. applying global message filters locally to reduce both network traffic and protocol message translations.

In this paper, we look at an approach to address these heterogeneity challenges. The OverStar software framework supports the generation of overlay-based middleware services from high-level declarative specifications; in particular it concentrates on supporting the specifications that achieve interoperability and optimisation of heterogeneous systems. For this purpose two separate model specifications are provided:

- *Overlay specification.* Each heterogeneous middleware service is underpinned by a tailored overlay network. Timed automata are used to specify two aspects of the overlay's behaviour. First, how the overlay topology should best be constructed to integrate the individual end systems (e.g. a tree, ring, etc.). Second, timed automata are also used to model the communication of data in the overlay network, e.g., multicast, anycast, etc.
- *Node behaviour specification.* Each overlay node acts as a gateway to the behaviour of the heterogeneous protocols in the local end systems. Protocol transparent middleware behaviour is then specified at each node to achieve interoperability and/or optimise service functionality. Such behaviour is specified using a timed automaton and can contain operations including: message translation, and message filtering.

We evaluate OverStar using a case study based method involving two middleware services in given areas of application: resource discovery and an event service. We show that these services can be specified and optimised in the face of heterogeneous protocols across the end-systems; interoperability can be achieved; and node behaviour specification supports the optimisations of deployments despite the encountered heterogeneity.

The paper is structured as follows. In section 2 we introduce the OverStar approach and associated software framework. In section 3 we then define the formal models that underpin the solution, and in 4 we describe the implementation of the OverStar framework. The evaluation results are given in section 5. In section 6, we analyse the work with respect to the state of the art. Finally, we draw conclusions in Section 7.

[3] http://stomp.github.com/
[4] http://activemq.apache.org/openwire.html

2 The OverStar approach

2.1 Motivation

We use a simple example to motivate the OverStar approach. Fig. 1 illustrates a set of end-systems that employ heterogeneous protocols to provide middleware services in their local domains. Multiple discovery protocols are shown in use: SLP, Bonjour, UPnP and Ariadne [11]. Similarly multiple event communication middleware: Active MQ brokers, XMPP XEP-0060 [5] and a publish-subscribe sensor middleware [24]. These can be viewed as isolated islands of interoperability that must be carefully integrated to create resource discovery and event services respectively. Further, in dynamic systems it is unrealistic to predict which end-systems protocols will be employed.

Fig. 1. Heterogeneous middleware services in systems-of-systems

Building global, optimised middleware services across heterogeneous end systems requires a substantial understanding of: distributed algorithms, different communication patterns, interoperability challenges and low-level network programming. Furthermore, different strategies are required for different contexts (e.g. the solution for resource discovery is different from an event service solution). The potential heterogeneity means that the solution space may rapidly grow, such that a single middleware solution is not sufficient. Hence, we argue that it should be possible to specify the middleware service optimised for the given context and then use this to generate the deployable middleware software.

2.2 The OverStar Middleware Framework

OverStar is a software framework that composes middleware services across heterogeneous end systems as illustrated in Fig. 2. *OverStar node instances* are deployed in multiple domains to communicate with heterogeneous end systems in order to underpin higher-level middleware services. A node instance is made up of two component types. First, the OverStar nodes must be globally connected in order to facilitate the optimised communication between heterogeneous domains. Overlay networks offer a well established solution for building

[5] http://xmpp.org/extensions/xep-0060.html

such virtualised services. However, a single overlay network is not sufficient; for instance, the overlay behaviour required to underpin a resource discovery service may differ from one to underpin publish-subscribe. Hence, the *Overlay Node component* allows different implementations of self-managing overlay behaviour to be created and deployed, e.g., a multicast tree or a DHT ring. Second, service implementation behaviour must be layered atop this overlay; this should connect the legacy systems in such a way that end-to-end interoperability is achieved. Further the service behaviour should be tailored for optimisations, i.e., adding specific service optimisation in spite of the heterogeneity, e.g. applying global publish subscribe filters in the end-system domains. The *Service Node component* allows different behaviours for specific middleware service implementations to be deployed. Finally, each node performs network communication with legacy middleware systems using *data ports*, and with the overlay via the *overlay port*.

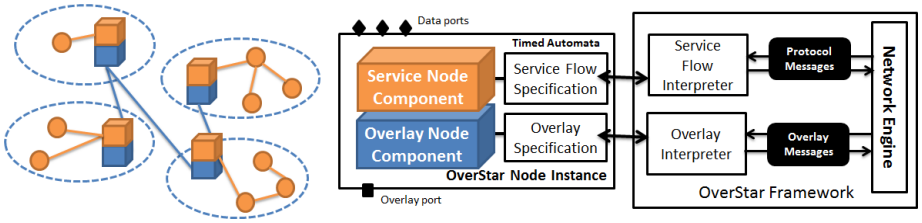

Fig. 2. An overview of the OverStar approach

In order to support the development of services and promote software reuse, the service and overlay node component's behaviour is specified through the use of timed automata (rather than hand-coded), which are finite state automata with a set of clocks, clock constraints, abstract messages, message constraints, queues and actions. We introduce the formal definition of these timed automata in Section 3.1, which are then applied to Overlay Node specifications in 3.2 and Service Node specifications in 3.3. We argue that the use of timed automata fits well with the requirements of global middleware services due to time constrained behaviour, e.g. in the self management of both overlay topologies and the middleware service logic.

The OverStar framework executes on each host and acts as an execution environment for the OverStar node instances. The elements of this framework (illustrated in Fig. 2) behave as follows:

– The *Service Flow Interpreter* interprets the internal model of a Service Node component to achieve the middleware service behaviour, e.g. supporting interoperability. Based upon the timed automata specification the interpreter: i) communicates with end system nodes using their legacy protocols, ii) communicates with other OverStar nodes using an Overlay Node component, and iii) performs middleware logic on the messages received from both.

- The *Overlay Interpreter* interprets the behavior specification of the Overlay Node component and performs the required local node behaviour to construct and maintain the overlay topology, and provide data communication services. That is, react to: join, leave, fail, and data events.
- The *Network Engine* performs two roles to support the two port types. Firstly, it sends the overlay specific messages between nodes in the overlay, i.e., these messages contain the overlay action messages, or forwarded data messages. Secondly, it communicates directly with legacy systems, sending and receiving messages using the required legacy protocol, e.g., it can send and receive SLP messages on the IP multicast channel of SLP.

3 Definition of Models to Specify Component Behaviour

Here we first present the formal definition of the timed automata used in OverStar. We then present the timed automata models for the Overlay node component specifications, followed by the Service node component specifications.

3.1 Timed automata specifications

Modeling time dependent behavior. Constraints on clock variables are used to model time dependent behavior. Local clocks are initialized to zero when a node starts and then increase synchronously with the same rate. Clocks associated to transitions act as *clock guards* that restrict the behavior of the automaton. Transitions from one state to another may not be taken according to time constraints, i.e. if a clock guard is not evaluated to true. Clocks may be reset to zero when a transition is taken. Further, to enforce progress properties, i.e. to ensure that nodes do not stay in a state forever, a state may be also associated with a clock constraint, called thereafter, *a local invariant*. For instance, as depicted in Fig. 3,❷, the local invariant ($x < 20$) associated to state s_1 ensures that the transition from state s_1 to s_5 is only taken if clock x has elapsed (i.e. is evaluated to more than 20 time units). In other terms, a local invariant determines how long an automaton can wait in a particular state for an event to be triggered. If the time expires and there is no transition satisfying the guards, then a violation of the constraints of the system occurs. More precisely, it means that there is a fault in either the specification of guards or invariants in the model; this is what is usually called a timelock. Such locks in OverStar specifications can be avoided thanks to the use of a timelock checker provided by timed automaton analysis tools. Additionally, a state is *urgent* when it has an invariant $x < 0$, with all its incoming transitions resetting x to zero. Hence, in an urgent state, the outgoing transition must be taken immediately (See Fig. 3,❸,❹,❺).

Abstract messages. Triggered events are messages received or sent from either i) the global overlay network referred to as an overlay port, or ii) end systems within a local domain, referred to as one or more data ports; these utilise legacy protocols e.g. SLP, XMPP, etc. for communication. Syntactical description of message data fields, including their data types are formalized through the use

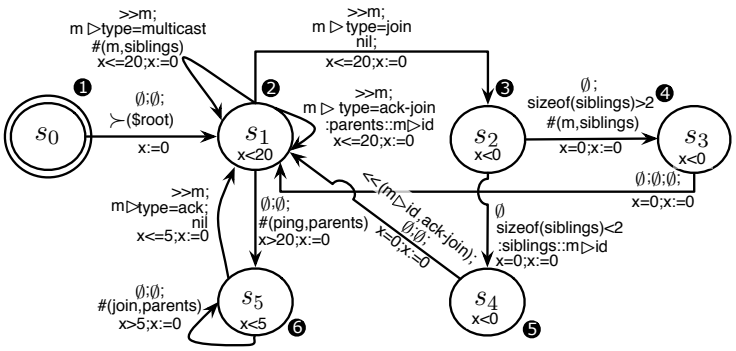

Fig. 3. Timed automaton specifying the self-organizing behavior of an overlay

of abstract messages [4]. An abstract message consists of a set of fields, either primitive or structured. The former is composed of: (i) a label naming the field, (ii) a type describing the type of the data content, (iii) a length defining the length in bits of the field, and (iv) the value, i.e., the content of the field. A structured field is composed of multiple primitive fields. We note $msg \triangleright field$ the operation that selects the $field$ from the abstract message msg. This abstract representation supports the application of additional message logic (e.g. message filtering) irrespective of the concrete packet format of a message.

Message guards. Transitions may also be labeled with a message guard that specifies a set of conjunctions of constraints on triggered events that has the following form: $(msg \triangleright field) \sim rvalue$ with $\sim \in \{<, \leq, =, \geq, >\}$ to evaluate adequately the message field $field$. As a result, a transition from one state to another can be taken only if both its clock and message guards evaluate to true.

Global variables and queues. Message guards may be combined with constraints on either global variables and queues. Both of them are accessible whatever the current state of a timed automaton. Global variables are variables prefixed with the '$' sign whereas queues are variables prefixed with the ':' sign. The term global refers to the states on one node, variables are not global to the distributed system. At any time, a timed automaton is able to store both incoming or outgoing messages to further get them back later.

Actions. When a state is left, actions may be triggered according to the transition to be taken. Available actions are described in Fig. 4 and include forwarding, multicasting, translating, filtering and queuing messages. We are providing a set of key actions to build overlay-based middleware services and provide end-to-end interoperability; however, the set of actions is extensible according to the needs of additional middleware functionality.

Formally, a timed automaton is defined as follows.

Definition 1. *A timed automaton $\mathcal{TA}$ is a tuple $(Q, M, q_0, \mathcal{A}, Evt, \mathcal{C}, Act, \mathcal{V}, \mathcal{F}, \rightarrow, \mathcal{I})$, where Q is a finite set of states, M is a finite set of abstract messages, $q_0 \in Q$ is the starting state and $\mathcal{A} \subset Q$ is a set of accepting states. Evt is*

Actions	
δm	Translate message m to $f(m)$
Filter(m, $fr_1...fr_2$)	Filter message m according to the field content filters $fr_1...fr_2$
$\propto m$	Multicast to the overlay network nodes m
$\succ$ \$*root*	Bootstrap procedure
$: q :: m$	Queue message m in queue q
$\#(m,id_1...id_2)$	Forward message m to overlay nodes $id_1...id_2$
λm	Multicast message m to the local environment

Fig. 4. Available actions in the model

a set of event types such that $Evt = \{?, !, \gg, \ll\}$ *where* ? *(resp.* !*) denotes a received (resp. sent) event from a data port, whereas* $\gg$ *(resp.* $\ll$*) denotes a received (resp. sent) event from an overlay port.* $\mathcal{C}$ *is a finite set of non negative real valued clocks and* $\mathcal{B}(\mathcal{C})$ *is the set of all clock constraints on* $\mathcal{C}$*. Act is the set of actions performed when a transition is taken.* $nil \in Act$ *is an empty action. The set of global variables and queues is respectively* $\mathcal{V}$ *and* $\mathcal{F}$*. Additionally,* $\mathcal{B}(M, \mathcal{V}, \mathcal{F})$ *is the set of constraint conjunctions on* M*,* $\mathcal{V}$ *and* $\mathcal{F}$*. Further,* $\rightarrow \subseteq Q \times Evt \times M \times \mathcal{B}(M, \mathcal{V}, \mathcal{F}) \times Act \times \mathcal{B}(\mathcal{C}) \times Q$ *is the set of transitions. Finally,* $\mathcal{I} : Q \rightarrow \mathcal{B}(\mathcal{C})$ *assigns local invariants to states.*

Concretely, transitions have the following form $s_1 \xrightarrow{\mathcal{L}} s_2$ and changes the state of timed automaton from s_1 to s_2 once the label $\mathcal{L}$ is evaluated to true. The transition label $\mathcal{L}$ is defined such as $\mathcal{L} \subseteq Evt \times M \times \mathcal{B}(M, \mathcal{V}, \mathcal{F}) \times Act \times \mathcal{B}(\mathcal{C})$, and has the following format:

$$\mathcal{L} = Event|Msg|Data\_guard|Actions|Clock\_guards$$

Correspondingly, four different transitions can be triggered according to events that can occur and are noted as follow (without considering guards and action for the sake of clarity): (i) $s_1 \xrightarrow{?m} s_2$ (resp. $s_1 \xrightarrow{!m} s_2$) if a message m has been received (resp. sent) from a local legacy system, (ii) $s_1 \xrightarrow{\gg m} s_2$ (resp. $s_1 \xrightarrow{\ll (id,m)} s_2$) if a message m has been received (resp. sent to id node) from the underlying overlay network. Further, our model also supports epsilon transitions. However, to avoid non-deterministic timed automata, such transitions must be combined with guards to avoid undeterminism (See for instance Fig. 3, ❸, ❹, ❺). It is important to note that epsilon transitions are only triggered when either timeout occurs or the current state is an urgent state.

3.2 Overlay Specification: Timed Automata to Construct Overlays

The first step in building a middleware service that integrates multiple legacy end systems is to construct the overlay topology and communication services that join them in a manner that the middleware functionality can be layered atop. Using the timed automaton definition, we are able to specify the algorithm to create such an overlay. To ease its understanding, Fig. 3 illustrates the

specification of a timed automaton $\mathcal{TA}_{tree}$ deployed at each node to create a self-managing tree overlay.

In particular, $\mathcal{TA}_{tree}$ is an instance of a timed automaton with a global variable $\$root$, two queues named *:parents* and *:siblings*, a clock x and three possible actions: $bootstrap, forward$ and $queue$ (respectively noted $\succ$, # and ::). An overlay node always starts with a *bootstrap* action that initializes both the $\$root$ variable and the clock variable x. The former variable is used to know if an overlay node is or is not the root of the tree based overlay whereas the latter variable is used to control the time dependent behavior (Fig. 3, vertex ❶). Then, overlay nodes must wait at most 20 time units for receiving either: (i) a multicasted data message, (ii) a join request, or (iii) a join acknowledgment (vertex ❷). In the first case, multicasted messages are forwarded to siblings of the overlay node. In the second case, according to size of the siblings queue, the node that has sent the join request may be added or not to the *siblings* queues of the current overlay node (vertex ❸). If it is added then an acknowledgment is sent to the requester (vertex ❺), otherwise the join request is forwarded to the siblings of the current overlay node (vertex ❹). Constraints on the size of the siblings queue enables avoidance of an unbalanced tree. In the third case, the node that receives a join acknowledgment adds the ack sender to its *parents* queue. To ensure that the overlay being built remains connected, each node must probe the liveness of its neighbours. Thus, beyond a delay of 20 time units, if no messages have been received, overlay nodes must poll their parent nodes to check if they are still alive (vertex ❻). If in less than 5 time units, no acknowledgment is received, overlay nodes may have been disconnected from the overlay and thus have to reforward a join request to their parents. Otherwise, if acknowledgments are received, overlay nodes go back to the listening state s_1 to receive messages.

The use of timed automata enables us to specify a fine grained overlay construction algorithm. In particular, it becomes easy to express timed dependent behaviors to perform overlay maintenance, to manage network errors, or to periodically check invariants of the overlay.

3.3 Sevice Specification: end-to-end middleware services

Overlay Nodes. At each node in the constructed overlay additional logic is deployed to perform the required middleware functionality that achieves a particular service. Specifically, this logic performs actions on messages received from either the local end systems, or from messages disseminated by the overlay network. As depicted in Fig. 5, all overlay nodes have an *overlay port* through which they can send messages to, and receives messages from other nodes in the overlay (dependent on the network service provided by the overlay, e.g. multicast). Each node also has a set of N *data ports* through which the node communicates with the end systems using the required middleware protocol, e.g. an SLP data port allows the overlay to communicate with end systems using this protocol.

Middleware functionality is then performed as *actions* on the messages received and exchanged between these ports; examples including message translation and filtering are defined in Fig. 4). We use simple examples to then illustrate

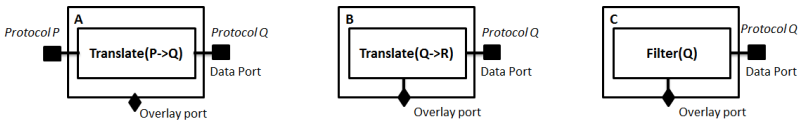

Fig. 5. Middleware logic actions applied at end system nodes

this procedure. Fig. 5A shows that a message received as a *Protocol P* message from a data port is translated to a *Protocol Q* message and sent on the corresponding data port (hence in this example the message is not transmitted to the overlay). Fig. 5B describes similar functionality but this time the message from the data port is translated before it is sent to the overlay. Finally, Fig. 5C illustrates the filtering of messages between the data ports and the overlay port. As previously stated, the approach is extensible to add new message actions to underpin a wider range of middleware services.

Data Flow Specification. Actions themselves are not enough to achieve service functionality; control logic is required to define the flow of data at the individual nodes. Thus, we further employ timed automata to specify the message flow across the overlay; a sequence of middleware, translating, multicast and queuing actions (resp. noted δ, $\propto$ and ::) are constrained by both time and message guards. Hence, it is possible to define different service functionality. For instance, in the case of a global resource discovery overlay integrating heterogeneous end system protocols, one solution is to follow a *translate and multicast strategy*: each node performs local translations between the disparate protocols employed in its domain; if there is no local resource match then the overlay node can pass the request to its neighbours by sending them the received incoming requests and/or their translated forms to increase chances to get successful answers. Fig. 6, illustrates a specification of a timed automaton $\mathcal{TA}_{bonjour1}$ that applies the aforementioned strategy to the *Bonjour* service discovery protocol. As soon as a message m of type `DNS_Question` is received, it is translated locally according to the underlaying gateway capabilities, to either SLP, UPnP, or other service discovery protocols (and noted $f(m)$) (Fig. 6, ❶). If `DNS_Response` messages are received locally (i.e. from the local environment) in less than 4 time units, they are sent to the requester (vertex ❺, ❻). Otherwise, the message m and its translated form $f(m)$ are queued and multicasted to overlay neighbors (vertex ❷).

Every 5 time units, if a `DNS_Response` message is received it is queued to be sent later (vertex ❸). Any other messages received are translated to *Bonjour* (vertex ❼). If the translation is successful, then it means that a `DNS_Response` has been received and then is queued. If the translation is not successful, the message $f(m)$ is either discarded if it has been already seen by the current node, or multicasted to overlay neighbors otherwise (vertex ❹). Finally, if the delay of 10 time units has passed without receiving any messages then all previously queued `DNS_Response` responses are sent to the requester, and the queue is flushed (vertex ❺, ❻).

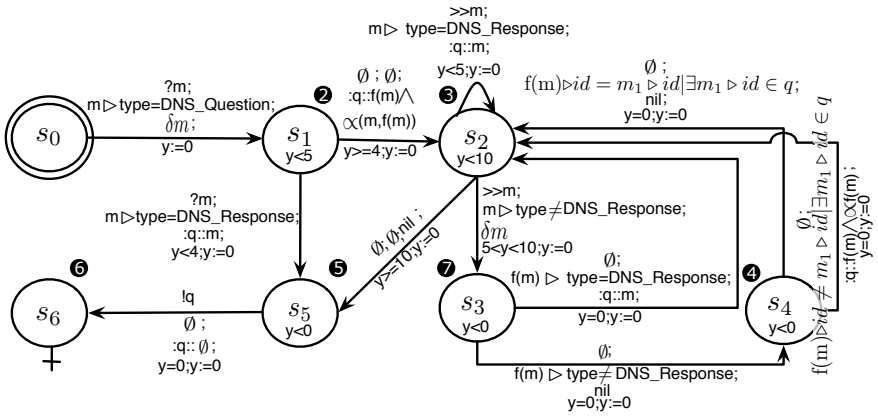

Fig. 6. Translate and forward strategy

From our model it is straightforward to define additional compelling strategies. For instance, in the case where the response time of the global resource discovery service is important, Fig. 6 can be altered to multicast incoming request immediately to find corresponding responses in other networked environment, without waiting for local responses. Further, Fig. 7 shows the node logic for event filtering in a global event service. Here, subscription requests are multicast across the network and translated and applied as local filtering rules. Published events are then translated to an abstract message specification to which the filters are applied. Published Messages that match the filters are translated to the legacy end system protocols and multicasted across either the local network or the overlay according to the messages' origin. This is a relatively simple publish-subscribe service that handles protocol heterogeneity; there are many potential broker strategies, which we believe the overlay and flow specifications are flexible enough to define.

Reusing overlays for multiple middleware services. The behavior of an overlay, noted $\mathcal{TA}_{\mathcal{O}}$, is modeled through a set of timed automata that are composed together. In a way similar to process algebras such as CCS [18] and FSP [17], we introduce the parallel composition operator $\|$ to compose timed automata. Hence, the behavior of $\mathcal{TA}_{\mathcal{O}}$ consists of individual timed automata that execute their transitions independently. As in our model, each timed automaton is independent from each other, compared to traditional process algebras, our composition operator $\|$ does not provide any synchronization features among composed timed automaton. Further clocks are local to each composed automaton. There are no global or shared clocks variable. So, provisioning n applications using P_1, P_2,..., P_n protocols across an overlay $\mathcal{O}$ is described by the following formula: $\mathcal{TA}_{\mathcal{O}} = \mathcal{TA}_{topology} \| \mathcal{TA}_{P_1} \| \mathcal{TA}_{P_2} \| \mathcal{TA}_{P_3} \| ... \| \mathcal{TA}_{P_n}$. The timed automaton $\mathcal{TA}_{topology}$ describes the self-organization behavior of the overlay and $\mathcal{TA}_{P_1}$, ..., $\mathcal{TA}_{P_n}$ specify the different translation strategy for each supported middleware service. The strength of our model comes from its flexi-

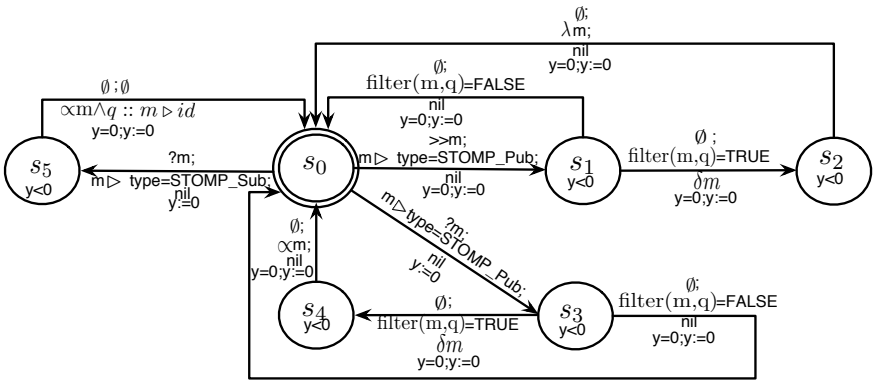

Fig. 7. Event translation and filtering strategy

bility. Replacing one strategy by another or taking into account a new service and/or a new overlay topology is straightforward as the $\|$ operator enables a modularized specification.

4 The OverStar Framework Implementation

Here we describe the further implementation details of the OverStar software framework. OverStar is implemented in Java and leverages the capabilities of the Starlink framework [4]. There are two key elements to the implementation: i) the implementation of reusable building blocks that underpin the *action* keywords that are performed during the service flow specification logic (e.g. interoperability), and ii) the implementation of the timed automata interpreters.

4.1 Actions: reusable software building blocks

As previously described, *Actions* are performed to realise the service flow logic behaviour. These are defined as key words in the timed automaton; and these key words relate to reusable software building blocks. Hence, the logic is extensible and adaptable through the creation of new building blocks.

Actions are specified using the Starlink framework. Starlink uses *k-colored automata* to capture the properties of a protocol by a color k and ensures that the messages are sent and received using the appropriate network service. This supports the parsing of a message into the *abstract message* format such that additional logic can be performed on the messages irrespective of the heterogeneous protocols. Hence, an action is a k-colored automata with the message logic relating to the action. When the timed-automata specifies a transition with a particular keyword action then the corresponding coloured automata is executed. To illustrate this method, we present one example in Fig. 8; this is a translation from SLP request messages to UPnP request messages. The original SLP

request message is translated to a SSDP request that initiates UPnP behaviour. The bi-coloured state performs the assignment of field data from one message to another. Other examples are: the translation of STOMP to XMPP messages, or the parsing of either of these such that they can be filtered by message topic.

Fig. 8. Starlink merged automaton for SLP to UPnP protocol translation

4.2 Timed Automata Interpreters

Both the Overlay and Service flow interpreters dynamically execute timed automata written in XML (we do not provide a schema here, however, the notation provided in Section 3 offers a concise representation). To illustrate how OverStar operates, we now summarise the behaviour that occurs at the two state types.

At a **receiving state**, the interpreter listens for messages from the overlay or data port. The receiving state parses the message to determine the automaton action e.g. translate, filter, etc. Transitions to other states are taken based upon both action types and guard conditions. For example, where there is a time-guard on the state a timeout exception is used, i.e. the state listens for new events and the timeout value is set to the guard value. If no event is received in the timeframe the exception is caught and the appropriate transition is executed.

At a **sending state**, the interpreter constructs a new instance of an overlay message to be forwarded in the overlay. This consists of the original legacy protocol message with a new OverStar header. The header contains a small amount of data (17 bytes) capturing the message type (e.g. forward, join in 1 byte), a unique message identifier (8 bytes), message source IP (4 bytes), and message source port (4 bytes). The sending state can also send concrete protocol messages to a given legacy end system using the correct protocol behaviour.

5 Evaluation

5.1 Case Study based Methodology

We employ a case-study approach to evaluate the ability of the OverStar framework to achieve its primary contributions. For this we developed two different but complimentary middleware services to highlight the flexibility of OverStar:

– A resource discovery service that can react to requests from heterogeneous end system protocols (e.g. SLP, Bonjour and UPnP) and ensure that matching service responses are returned.

– An event service that joins end systems using heterogeneous publish-subscribe technologies, e.g., STOMP and XMPP-XE0060 and ensures that events that match subscriptions are received despite the heterogeneous protocols.

The two services employ very different legacy middleware technologies and pose different challenges to applicability of OverStar. The resource discovery and event service solutions were deployed in the emulated complex network environment as described in Section 5.2. Utilising this experimental setup, we performed three measures: i) the end-to-end interoperability achieved by OverStar; ii) specific optimisations within the two services as specified by the service logic; and iii) the overheads occurred during OverStar's operation. These results are used to evaluate the extent to which the primary contributions are achieved.

5.2 Experimental Setup

To evaluate various aspects of OverStar, we have setup a particular network environment enabling reasonably large scale experiments. We have deployed OverStar across heterogeneous domains (e.g. 4, 8, 16) interconnected *via* a network backbone. A heterogeneous domain is instantiated as a Virtual Local Area Network (VLAN). A VLAN contains a set of devices that are logically connected within a single broadcast domain, and located in the same IP subnet. In fact, a one-to-one mapping between VLANs and IP subnets is applied, according to the best practices in network design. Devices may host either the OverStar middleware to act as an OverStar node, or middleware services relying on heterogeneous protocols. The key advantage of using VLANs to interconnect devices is to confine traffic generated by services (e.g. broadcast, multicast and/or unicast) into one domain without interfering with another, while abstracting the underlying physical network topology. Additionally, in our experiments, devices are emulated *via* Linux Kernel-based Virtual Machines (KVM) to use real operating systems and run unmodified both middleware services and OverStar middleware. The whole setup was conducted on a rack server equipped with 4 AMD opteron processors at 2 GHz, including 12-core per processor (for a total of 48 cores), and 32 GB of RAM. The server multiplexes virtual resources such as VLANs, KVMs on top of physical ones, and enables IP routing between domains.

5.3 Interoperability Experiments

In the emulated environment, we deployed a set of heterogeneous end systems across different domain configurations, i.e., four domains, eight domains, and sixteen domains; where in each domain, heterogeneous end systems utilise one of: SLP, Bonjour and UPnP to request or advertise a resource. We then specified and deployed an OverStar service solution using a multicast tree overlay timed-automata to connect the domains (up to sixteen). The service was specified to immediately multicast received requests from the heterogeneous end-systems onto the overlay; when received at the domain nodes these are translated to perform discovery using the local protocols. We measured the number of successful

match responses to the requests as the percentage interoperability achieved; this was compared to: i) no interoperability solution deployed, and ii) local bridges (i.e. bridges for SLP to UPnP, Bonjour to SLP, etc. deployed in each domain). The results in Fig. 9 show that local bridges increase the potential interoperability as they reach more services in the local domain, but OverStar achieves the necessary end-to-end interoperability via the global integrated service (n.b. across the experiment there is a least one matching service, and in many cases multiple matches). A similar experiment was performed for heterogeneous end-system event services (STOMP and XMPP-XE0060). Here the OverStar specification used a multicast tree with the local filtering only timed-automata (see Fig. 7). To measure the interoperability percentage in this case we compared the actual received events as a percentage of the matching events published across the network. Similarly, OverStar is able to achieve end-to-end interoperability in the event service case compared to the local domain approaches. Overall, these results demonstrate that hypothesis one is proven.

Fig. 9. Percentage interoperability results

5.4 Optimisation Experiments

For the optimisation experiments we use the same experimental setup as with the previous experiment. However, this time we apply different timed-automata strategies for the middleware service specifications. For the resource discovery case: i) multicast and translate (as used in the interoperability experiment), and ii) match service requests locally and multicast to the overlay when there isn't a response (this specification is captured in Fig. 6). For the event service case: i) local filtering only (where all publications are multicast on the overlay and local

filters determine which are translated to the end-systems (see Fig. 7), and ii) global filtering where subscriptions are multicast to the OverStar nodes to ensure that only matching publications are sent between domains. The results in Fig. 10 show that for resource discovery, strategy one reduces the maximum response time from a matched service in the global network, but this approach occurs significant message overhead especially as the domain configuration grows larger. Strategy two reduces this number of messages sent in the network, although the maximum response time is increased. It is interesting to note that the deployment of local bridges in a domain can create a cycle (i.e. the message is translated from one protocol using one bridge and then back to the same protocol by a separate bridge) leading to an infinite number of messages in the domain and across the network. The use of OverStar is shown to prevent such cycles occurring. Finally, in the case of the strategies for the event service, it can be seen that strategy two reduces the overall number of messages in the network compared to strategy one; hence, this minimises the message translations that take place. Overall, these results show that OverStar can be flexibly used to optimise for different domain configurations and requirements and offer initial proof of hypothesis two.

	Resource Discovery						Event Service	
	Strategy One			Strategy Two			Strategy One	Strategy Two
	No.	Min resp.	Max resp.	No.	Min resp.	Max resp.	No.	No.
Domains	msgs	(ms)	(ms)	msgs	(ms)	(ms)	msgs	msgs
4	1593	647	1454	833	672	4673	1372	441
8	3012	638	1625	1343	723	4836	2618	629
16	6206	655	1595	2721	756	4793	453	1074

Fig. 10. Comparison of different service strategies

5.5 Resource Overheads Experiments

Finally, we examine the resource overheads of the OverStar implementation. For this, we measure the time taken to perform three indicative individual actions on each OverStar node: i) the time to translate from an SLP message to a Bonjour message, ii) the time to translate from a STOMP message to an XMPP message, and iii) the time to translate from a STOMP message to an abstract message and then perform filtering. The results in Table. 1 show that OverStar introduces an expected overhead, however, this does not detract significantly from the overall performance of the services (e.g. compared to the overall response time of resource discovery). N.b. the measures are dependent on the protocol types; STOMP to XMPP involves text to XML message translation and hence is slower than SLP to Bonjour which is a binary to binary translation.

Action	Time (ms)
SLP to Bonjour	0.28
STOMP to XMPP	0.39
STOMP to filter	0.25

Table 1. Direct bridging deployments only

6 Related Work

6.1 Interoperability Solutions

Interoperability solutions focus on the search for a universal standard; and where such a standard is agreed and adopted the problem is solved. However, history has shown this approach to be unsuccessful. Two primary examples: the set of CORBA standards from the OMG [10] and the set of Web Services standards [3] from the W3C. However, such one size fits all standards are not suited to the extreme heterogeneity of systems-of-systems, e.g. from small scale sensor applications and embedded devices through to large scale Internet applications.

Rather than seek universal standards, alternative approaches either build direct bridges between systems e.g. the SOAP to CORBA bridge [6], embrace simplicity (i.e. RESTful solutions) or look for transparency (i.e. Service Buses). REST presents a simple uniform API atop a global standard protocol (the HTTP protocol being widely used to connect systems) allowing many interoperability problems at the communication level to be addressed. However, the Restful approach leaves interoperability issues arising at the application behaviour and data level unresolved. For example, a service cannot respond to a GET operation request composed of an operation name and data parameters that has different behaviour and syntax to itself. Opposed to standards, transparent solutions mimic the action of a language interpreter, that is they receive communications from system A and then translate this such that system B can understand and vice versa. Enterprise Service Buses (ESBs) e.g. Artix, INDISS [5], z2z [6], Janus [1] and uMiddle [19], provide such capabilities between multiple "languages". However, transparent solutions are typically restricted to a set of known middleware types, and the development effort required to extend them for new protocols is significant. CONNECT [2] has examined semantics-based solutions to automate this challenging task, however, the focus is single party protocols between two systems; within the CONNECT approach, [13] examines interoperability between heterogeneous multiparty middleware abstractions, but does not consider the underlying deployment complexities of achieving end-to-end interoperability in heterogeneous systems-of-systems.

Analysis. Generally, interoperability has been considered from an enterprise systems perspective, where interactions are point-to-point, planned and long-lived. Hence, they remain limited when considering the dynamic composition of heterogeneous systems, where the knowledge about the services provided by

[6] http://soap2corba.sourceforge.net/

different systems and the protocols they employ are unknown until binding time and a common translation technology cannot be agreed upon in advance.

Further, none of the above solutions considers the cases of interoperation between systems using heterogeneous multi-party communication protocols (e.g. multi-party discovery, group communication, publish subscribe, etc.), they consider only the case where a single system must interact with another. OSDA [15], MUSDAC [20] and SeDiM [7] offer bridging solutions between service discovery domains to provide universal solutions i.e. a service lookup request from one domain can be answered in another network domain irrespective of the service discovery protocols employed in that domain. Notably, OSDA uses a peer-to-peer ring to communicate messages between heterogeneous domains. However, the weakness of these platforms are threefold: i) they are specific solutions implemented for service discovery and cannot be flexibly applied to other problem domains e.g. group communication; ii) they employ a transparent intermediary between domains and hence mappers to and from this intermediary must be developed by hand for every protocol, and iii) the intermediary is a 'subset of all protocols' and as such this subset may become too small to underpin interoperability in a general fashion, e.g., if service discovery protocols A and B provide attribute based lookup while protocol C does not then the intermediary cannot include attribute lookup; this lessens any potential interoperation between A and B. In comparison, OverStar supports the specification of end-to-end interoperability solutions between heterogeneous multi-party middleware protocols that span heterogeneous network domains.

6.2 Overlay Networks and Middleware

Overlay networks are virtual communications structures that are logically 'laid over' an underlying physical network. They are established solutions for providing scalable application services across heterogeneous networks, nodes and systems. For example, publish-subscribe and group interaction can be underpinned in the Internet by multicast overlays such as SRM [8]. Similarly, DHT-based peer-to-peer overlays provide reliable resource discovery in large-scale distributed systems e.g. Pastry [22] and Chord [25]. And publish subscribe services are one example of middleware services layered atop DHT, e.g., Scribe [23]. These properties make them suited to connecting heterogeneous systems of systems; yet the different types of middleware protocols suggests that a single network type is insufficient and it must be possible to flexibly specify an overlay to underpin the broad range of potential middleware services.

There exist toolkits that provide principled support for overlay network development. *JXTA* [7] is a framework where p2p applications are developed atop a resource search abstraction; this supports grouping and contacting nodes. This abstraction can be implemented using a number of overlay topologies. This approach involves a full development life-cycle and hence, higher-level declarative

[7] http://www.jxta.org

languages and models have been produced to simplify the complex task of constructing new overlays. *Macedon* [21] is a state machine compiler for overlay protocol design. Event-driven state machines (EDSMs) have been used over decades for protocol design and specification. Macedon extends this approach to an overlay specific, C++ based language from which it generates source code for overlay maintenance and routing. In the P2/Overlog project [16], applications use a declarative logic language to specify their requirements of the overlay network. This is combined with a data flow approach, as opposed to a finite state machine approach, to maintain the overlay at runtime. Like Macedon, this simplifies the development process of overlays in specific cases. *iOverlay* [14] provides a message switch abstraction for the design of the local routing algorithm. The neighbors of a node are instantiated as local I/O queues between which the user provided implementation switches messages. This simplifies the design of overlay algorithms by hiding the lower networking levels.

Analysis. While suited to the construction and maintenance of overlay networks, the above are limited with respect to the high-level declaration and deployment of the atop application services. That is, it is not possible to specify the data-flow behaviour in terms of handling the problem of end-to-end interoperability. In comparison, OverStar supports the declarative specification of middleware services atop overlay networks in order to optimise the flow of message data and the necessary dynamic translations between protocols .

7 Concluding Remarks and Future Work

In this paper we have highlighted the importance of integration of end systems leveraging heterogeneous middleware; and here, end-to-end interoperability is a key requirement. Indeed, it can no longer be assumed that a single protocol is used across network and organizational boundaries in order to implement network services such as service discovery, multicast, group communication and publish subscribe. Instead, heterogeneous protocols will be employed. In the face of this heterogeneity, new approaches to build global middleware services are required that ensure that all services and devices are connected in an efficient and optimised way in order to effectively coordinate.

For this purpose, we have introduced novel models that specify overlay behaviour to support the development of middleware services that achieve end-to-end interoperability in complex systems-of-systems and an associated software framework (OverStar). The key contributions of which are the use of timed automata for: i) the specification of the topology and maintenance of the overlay network which interconnects heterogeneous protocols across large-scale networks; ii) the specification of the overlay's application service, in this case the logic and flow tailored to the particular middleware service type. We evaluated this framework using both resource discovery and event communication services. Our initial results from the simple case-studies have shown that the OverStar solution increases interoperability within the network and reduces the resource consumption in terms of messages sent compared to bridging solutions.

There are a number of interesting avenues of future work. The first is to extend the models in order to capture improved strategies for performing optimised, scaleable, end-to-end interoperability of resource discovery, group communication, and publish subscribe services. In this regard, overlay networks are well suited to self-organizing behaviour, hence there is the potential for the overlay to monitor the environments and protocols in order to better determine how to optimise the deployed middleware service. The use of interpreted models provides a mechanism to easily adapt the behaviour of the service by dynamically changing the model at runtime. Complimentary to this, the use of machine understandable models, i.e., timed automata, makes machine learning of solutions an interesting way forward; for example, machine learning protocols have been used to learn the automata for individual network protocols [12], and there is the possibility of learning more complex overlay network specifications.

Acknowledgments.

This work is part funded by the CONNECT project, funded under the Framework 7 FET Programme: http://www.connect-forever.eu.

References

1. T. Bissyande, L. Reveillere, Y. Bromberg, J. Lawall, and G Muller. Bridging the gap between legacy services and web services. In *ACM/IFIP/USENIX 11th International Middleware Conference, Bangalore*, volume 6452 of *Lecture Notes in Computer Science*. Springer, 2010.
2. G. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: supporting interoperability in complex distributed systems. In *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, Middleware'11, pages 410–430, Berlin, Heidelberg, 2011. Springer-Verlag.
3. D. Booth, H. Haas, F. McCabe, E. Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture, February 2004.
4. Y.-D. Bromberg, P. Grace, and L. Reveillere. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *31st International Conference on Distributed Computing Systems (ICDCS-2011)*, pages 446–455, June 2011.
5. Y.-D. Bromberg and V. Issarny. Indiss: Interoperable discovery system for networked services. In *IFIP/ACM/Usenix International Middleware Conference*, pages 164–183, 2005.
6. Y.-D. Bromberg, L. Réveillère, J. L. Lawall, and G. Muller. Automatic generation of network protocol gateways. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 21–41, Urbana Champaign, IL, USA, 2009. Springer-Verlag New York, Inc.
7. C. Flores, G. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad hoc environments. *IEEE Distributed Systems Online*, 2007.
8. S Floyd, V. Jacobson, C. Liu, S. McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5:784–803, December 1997.

9. P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 123–136, New York, NY, USA, 2008. ACM.
10. Object Management Group. The common object request broker: Architecture and specification version 2.0. Technical report, 1995.
11. Yih-Chun H., A. Perrig, and D. Johnson. Ariadne: a secure on-demand routing protocol for ad hoc networks. *Wirel. Netw.*, 11(1-2):21–38, January 2005.
12. F. Howar, B. Jonsson, M. Merten, B. Steffen, and Sofia Cassel. On handling data in automata learning - considerations from the connect perspective. In *ISoLA (2)*, pages 221–235, 2010.
13. V. Issarny, A. Bennaceur, and Y. D. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer, 2011.
14. B. Li, J. Guo, and M. Wang. iOverlay: A Lightweight Middleware Infrastructure for Overlay Application Implementations. pages 135–154. 2004.
15. N. Limam, J. Ziembicki, R. Ahmed, Y. Iraqi, D. Li, R. Boutaba, and F. Cuervo. Osda: Open service discovery architecture for efficient cross-domain service provisioning. *Computer Communications*, 30(3):546–563, 2007.
16. B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 75–90, New York, NY, USA, 2005.
17. J. Magee and J. Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.
18. R. Milner. *Operational and Algebraic Semantics of Concurrent Processes*. 1990.
19. J. Nakazawa, H. Tokuda, W. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006.
20. P. Raverdy, V. Issarny, R. Chibout, and A. de La Chapelle. A multi-protocol approach to service discovery and access in pervasive environments. In *Mobile and Ubiquitous Systems - Workshops, 2006. 3rd Annual International Conference on*, pages 1 –9, july 2006.
21. A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and Amin Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *In NSDI*, pages 267–280, 2004.
22. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.
23. A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, pages 30–43, London, UK, UK, 2001. Springer-Verlag.
24. E. Souto, G. Guimar, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal Ubiquitous Comput.*, 10(1):37–44, December 2005.
25. I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11:17–32, February 2003.