The design of a generalised approach to the programming of systems of systems

Geoff Coulson, Gordon Blair, Yehia Elkhatib, Andreas Mauthe
School of Computing and Communications
Lancaster University
Lancaster, UK
geoff@comp.lancs.ac.uk

***Abstract*—The world's computing infrastructure is increasingly differentiating into autonomous systems (e.g. Internet of Things installations, clouds, VANETs, …), which are then post-hoc composed to generate value-added functionality ("systems of systems"). Today, however, such system-of-systems composition is typically carried out in an ad-hoc and system-dependent manner, with obvious associated disadvantages. In this paper, we propose a *generalised* system-of-systems-oriented programming approach that enables systems to be composed by application experts without the need for systems-level knowledge, and also facilitates dynamic and spontaneous system composition, as systems discover each other opportunistic in their environment.**

Keywords—systems of systems; IoT programming model

I. Introduction

The world's distributed computing environment is becoming increasingly diverse and differentiated in nature, to the extent that it is now far removed from the traditional picture of PCs/mobiles + IP networks. The picture today includes: Internet of Things installations like smart cities and buildings, environmental sensor and actuator networks (WSNs) using non-IP protocols, cluster-based cloud systems, ad-hoc networks such as MANETs and VANETs, and virtualised systems supported by network overlays. At the same time, these various "systems" are often interconnected, so they can interact with and respond to each other. For example, VANETs talk to smart cities, WSNs send data to back-end clouds for processing, and overlay-based systems need to maintain their resilience properties when their underlying IP environment changes.

This is all leading to a world-view in which we are increasingly concerned with *composing* systems to build *systems of systems* [1]. Although this fact is becoming recognised in many research communities, surprisingly little work has yet been done on *programming models/environments* to facilitate system-of-systems composition. Instead, system composition is typically carried out in an ad-hoc manner, reliant on detailed knowledge of the internals of the systems being composed. We argue that a generalised and principled programming approach is urgently needed to support the full fruition of the emerging system-of-systems world.

We acknowledge the support of the EU's CHIST-ERA framework which has funded the "DIONASYS: Declarative and Interoperable Overlay Networks, Applications to Systems of Systems" project (grant EP/M015734/1) under which this research has been carried out.

We are currently developing such a system-of-systems-oriented programming approach. A key aspect of our approach is to assume that systems interact and compose *opportunistically*. In this way, we see systems-of-systems arising as a result of spontaneously-arising, mutually-benefitting, time-bounded, alliances between autonomous systems that dynamically discover potential partner systems in their environment. Here are some familiar examples of such opportunistic interactions/compositions: i) on arrival, a networked team of rescue workers dynamically interfaces with a local hospital's systems; ii) passing vehicles exchange traffic information; iii) isolated islands of connectivity in a sparsely-populated area discover a delay-tolerant overlay network through which they can interact; and iv) layers on distinct systems (e.g. a phone and a laptop) compose opportunistically when they happen to come in range of each other. The general pattern is one of loosely-coupled interaction between self-contained systems, triggered opportunistically by relevant "contact" events (e.g. discovery of physical proximity).

Our programming approach, discussed in this paper, is based on a first-class programmatic abstraction of a "system", which we call a *tecton*[1]. *A tecton is a distributed representation of a potentially-opportunistically-interacting distributed system*. Tectons exhibit the following properties:

- *Generality*: tectons uniformly represent the full range of distributed "systems", whether infrastructure-level/user-level, network-core/network-periphery, wired/wireless, fixed/mobile, large/small, static/dynamic. Examples are: user groups, MANETs, clusters, clouds, overlays, VPNs, SDN domains, WSNs, or even (on a more 'micro' scale) individual devices that can plug-and-play with other devices (like a generalisation of Bluetooth). Tectons are also intended to model *virtual* systems that are defined dynamically using predicates – e.g. we might define a tecton in a WSN environment that includes all nodes with >90% reliability.
- *Opportunistic interaction*: tectons have the ability to interact when they make potentially-mutually-beneficial "contact" (e.g. through node mobility). This applies in both a *horizontal* and a *vertical* sense, involving

[1] The term "tecton" is derived from the geophysical notion of tectonic plates: the implication is that systems come into spontaneous horizontal contact, and may also 'slide' vertically over each other.

(respectively) interactions between peer systems at the same level, and cross-layer interactions.

- *Declarative, rule-based, programmability*: In both the horizontal and the vertical cases, opportunistic interaction is managed by providing tectons with programmatic *contact-action rules* that specify the conditions under which two tectons are deemed to have made mutually-beneficial contact, and determine the form of the consequent composition or interaction[2]. "Contact" is defined using arbitrary predicates; actual physical contact (e.g. through node mobility) is just a special case of this more generalised notion of contact.
- *Low overhead*: tectons need to be supportable at run-time with modest resource over-head: e.g. the runtime element of the abstraction should run on resource-poor wireless PDAs and sensor nodes.

Fig. 1 abstractly illustrates the notion of tectons. Four tectons are represented as round-edged rectangles. The internal dotted ovals are the underlying "real systems" that the enclosing tectons are representing. Similarly, the "real" system nodes are shown as black triangles, and the corresponding blue ovals are their representatives in the tecton world. The red dots represent the above-mentioned contact-action rules which manage "contact" with other tectons. Potential contact is illustrated in both vertical and horizontal planes. When mutually-beneficial contact is detected and announced, an "action" associated with that contact is initiated; this is illustrated by the curved blue arrow.

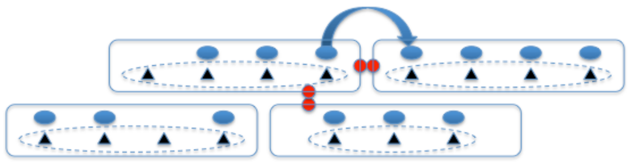

Fig. 1. An abstract view of tectons.

Contact-action rules are central to the dynamic/opportunistic nature of the model. For example, we might program a tecton representing a MANET by (essentially) saying "**IF** *any of the nodes in this MANET come into contact with a node of a tecton representing an 802.11 network,* **THEN** *all the MANET's nodes should reconnect to their email servers via 802.11*". Extrapolating from this, it can be seen how contact-action rules would enable the tecton programmer to specify, in a very general way, the conditions under which two systems should opportunistically interact, and what form that interaction should take. More broadly, we envisage the whole process of developing, deploying and managing systems of systems as amounting to a process of defining horizontally- and vertically-composable tectons along with their associated contact-action rules.

In this paper, we first, in Section II, outline the architecture of our proposed tecton-based programming system. Then, in Section III, we focus on one key foundational aspect of the architecture: the *tecton-machine runtime* that must be supported on each node that is able to participate within the tecton world. Finally, we briefly survey related work in Section IV, and offer conclusions in Section V.

[2] This form of interaction management echoes interactions between single-celled organisms and proteins: cell walls have sites which, when they come into contact with 'matching' proteins ("contact"), interact with them according to well-known physical and chemical laws ("action").

II. Tecton Architecture

As illustrated in Fig. 2, the architecture of our tecton-based programming approach comprises two layers, separated by the dotted line. Above the dotted line we have the "high-level" aspects of the architecture, as follows:

- *Domain-specific languages (DSLs)*: The desired generality of the tecton approach, in both technology and application domain terms, implies that a single set of programming language constructs is unlikely to be sufficient. Instead, we look to support multiple languages that capture concepts of relevance in different contexts. For example, a systems programmer working with SDN-based network-level tectons might expect a DSL based on C or Java; whereas a smart building manager working with tectons that represent things like "*all the window actuators on this floor*" or "*the group of today's visitors*" might favour a scripting or graphical approach.
- *Ontologies*: Similarly, the extreme variance in domains of application dictates that the information concepts available to the programmer will vary according to the domain of application. Our approach here is to employ ontologies [2] to structure these concepts. We only assume that is it possible ultimately to map ontology statements to sets of <*name*, *value*> pairs that can be processed by the low-level aspects of our architecture, as discussed below.

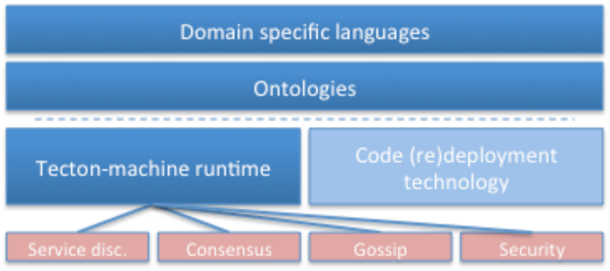

Fig. 2. The architecture of the tecton programming system.

Please note that this paper does *not* discuss these high-level aspects in any detail; instead it focuses on the low-level foundations of the approach, below the dotted line:

- *Tecton-machine runtime*: This is the heart of the tecton architecture; an instance of it is available on each device able potentially to participate in the tecton world. The API offered by the tecton-machine enables client code to carry out basic operations such as creating/destroying tectons, managing the membership of tectons (i.e. which nodes are members of which tectons), setting the criteria for "contact" when two tectons come into contact with each other, and coordinating consequent action (e.g. interaction, composition, etc.). The API provides low-level support for whatever ontologies are operating at the higher-levels in terms of <*name*, *value*> metadata annotations on tectons and nodes. The semantics/ranges/values of these are determined by the supervening ontology.

- *Code (re)deployment technology*: As suggested by its shaded colour, this is an "optional" aspect of the architecture. We view it as highly desirable for nodes to support dynamically-deployable software because the associated flexibility vastly increases the scope of the "action-consequent-to-contact" referred to above. For example, if two tectons representing network overlays come into contact and want to merge, this can be achieved by dynamically deploying mutual packet format converters into border nodes. However, we do not mandate code (re)deployment technology in our architecture, as this could unduly limit the applicability of our approach on primitive devices such as wireless sensors.

- *Plug-ins*: These are the architectural elements coloured red in Fig. 2. Tecton-machines can be configured with alternative implementations of *service discovery protocols* (SDP uPnP, Bluetooth, etc.], *election protocols* [4], [16], *gossip/epidemic protocols*, and *security protocols*. The ability to dynamically configure a tecton-machine with alternative plug-ins is dependent on the availability of code (re)deployment technology as discussed above. More detail is given in Section III.

III. The Tecton-Machine Runtime

We now define the basic concepts used by the tecton-machine runtime, present its API, and discuss how tecton-machine instances inter-work.

A. Tecton-machines and tecton-spaces

As mentioned, we assume that every device that can participate in the tecton world supports an instance of the tecton-machine runtime. In more detail, we assume an implementation substrate comprising sets of *tecton-machines* collected into *tecton-spaces*. Tecton-machines run on machines (either physical or virtual) that are assumed to support the "real" distributed systems being represented by tectons. Tecton-spaces are distributed environments that contain a number of tecton-machines, and provide them with connectivity. The expectation is that tecton-spaces correspond loosely to practical delineations in the real world: e.g., a university campus, an ISP, or a mountain rescue centre. As we will shortly see, there is no restriction that a given tecton needs to live within a single tecton-space: they can span multiple tecton-spaces as long as connectivity (even delay-tolerant connectivity) is available. In addition, tecton-spaces can be provided at different layers: for example a university campus might provide one tecton-space comprising all its routers and another containing all its end systems.

Fig. 3 illustrates the concepts of tecton-machines and tecton-spaces. It shows two tecton-spaces, one above and one below. Each tecton-space contains four inter-connected tecton-machines (blue trapezoids). The top tecton-space is supporting two tectons: the left one with four nodes, and the right one with two. Contact has apparently been announced between these two tectons, as the small curved blue arrow bridging them indicates action-consequent-to-contact. As indicated by the larger curved blue arrow, contact has also apparently been announced between tectons across tecton-space boundaries; this is discussed below. The straight arrows emanating from the rightmost tecton-machine in the top tecton-space represent probes from a service discovery plug-in. These probes seek tecton-machines in other tecton-spaces; and when one is discovered, the possibility is opened of contact between tectons in different tecton-spaces.

The tecton-machine runtime comprehends the notion of a "coordinator node" in each tecton that is responsible for coordinating decision making and action-consequent-to-contact. Coordinator nodes are illustrated in purple in the figure. To protect against these being vulnerable single points of failure, the tecton-machines in a tecton-space take collective responsibility for monitoring each tecton's coordinator node and electing a new one (using an election protocol plug-in) if the current one withdraws from the tecton, or its tecton-machine crashes.

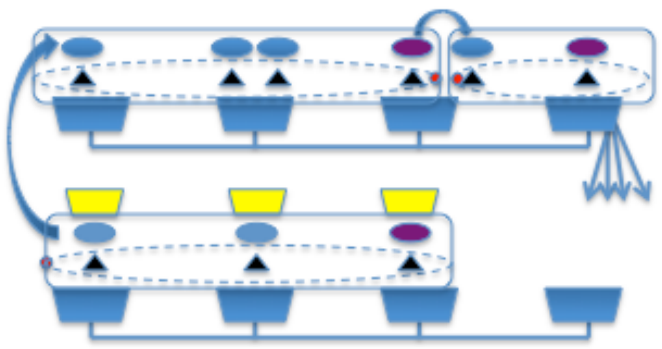
Fig. 3. Tecton-machines and tecton-spaces: 3 tectons in 2 tecton-spaces.

Finally, Fig. 3 illustrates the possibility of recursive application of the architecture: the three nodes in the bottom tecton are shown supporting an upper layer of tecton-machines (shown in yellow) that comprise a "nested" or virtualised tecton-space. This is useful in deployment environments that incorporate code (re)deployment technology, to enable the *migration* of tecton-spaces from one location to another.

B. Tecton-machine API

The API exported by each tecton-machine is divided into the following three categories: tecton lifecycle management, internal tecton management, and external tecton behaviour. We now expand on these (note that upcalls are shown <u>underlined</u>).

1) Tecton lifecycle management

```
tecton_id, node_id
Create(Metadata name_val_pairs,
       Predicate siteselectioncriteria,
       Handler code_ptr);
CreatedNotification(tecton_id tecton,
                    Metadata name_val_pairs);
Destroy(tecton_id tecton);
DestroyedNotification(tecton_id tecton);
```

The *Create()* operation creates a new tecton and an initial coordinator node for the tecton. The metadata embodied in the *name_val_pairs* argument are attached to the new tecton; as explained above, these metadata are assumed to be given a semantic by a supervening ontology. The *Predicate* argument determines which tecton-machines in the hosting tecton-space should be informed of the creation of the new tecton via an upcall. A predicate is represented as a sum-of-products boolean expression, defined over <*name*, *value*> metadata attached to each tecton-machine in the tecton-space. *Create()* returns a *tecton_id* which uniquely identifies the new tecton. It also returns a *node_id* that is made to correspond to coordinator node functionality provided by the client (the *code_ptr* argument) and thus identifies the tecton's coordinator node.

CreatedNotification() is an upcall that is delivered to client code of the tecton-machines that were selected by the *Predicate* argument to *Create()*. The <*name*, *value*> metadata that were passed to *Create()* are delivered to this client code which, as a consequence, may choose to register itself as a node of the new tecton using *AddNode()* (see below).

Destroy() and *DestroyedNotification()* work similarly to *Create()*/*CreatedNotification()*. *Destroy()* may only be called on the tecton-machine supporting the tecton's coordinator node. *DestroyedNotification()* is upcalled on all tecton-machines that had a node in the tecton.

2) *Internal tecton management*

```
node_id
AddNode(tecton_id tecton,
        Metadata name_val_pairs,
        Handler code_ptr);
SetTectonMetadata(node_id node,
                  Metadata name_val_pairs);
SetNodeMetadata(node_id node,
                Metadata name_val_pairs);
RemoveNode(tecton_id tecton, node_id node);
CoordNodeNotification(node_id new_coord);
```

Typically, *AddNode()* is called on receipt of a *CreatedNotification()* upcall. It is used to register a piece of client code as a tecton node and to associate initial <*name*, *value*> metadata with the node. It is permissible to add a node to a tecton supported by an external tecton-space, and thus to enable tectons to span tecton-space boundaries. Note that it is not always necessary to represent *every* node in the real underlying "system": only nodes involved in contact detection and/or action-consequent-to-contact need to be represented as tecton nodes. For example in some WSNs it may only be necessary to represent the cluster heads.

SetTectonMetadata() and *SetNodeMetadata()* associate new <*name*, *value*> metadata with the specified *tecton_id*/*node_id*. They are called by client code whenever aspects of the underlying "real system's" environment changes in ways likely to be of relevance to *SetContactCriteria()* (see below).

RemoveNode() is called to remove a node from a tecton. It may be called only from the tecton-machine that hosts the node. If the coordinator node gets removed, the tecton-machines in this tecton-space elect a new coordinator node for this tecton; and the "winning" tecton-machine uses *CoordNodeNotification()* to inform the chosen node that it has been elected.

3) *External tecton behaviour*

```
SetContactCriteria(tecton_id tecton,
                   Predicate contactcriteria);
ContactNotification(
        tecton_id this_tecton, tecton_id other_tecton,
        node_id_list involved_nodes_in_this_tect,
        Metadata_list name_val_pairs_this_tect,
        node_id_list involved_nodes_in_other_tect,
        Metadata_list name_val_pairs_in_other_tect);
Action(tecton_id tecton,
        Metadata name_val_pairs,
        Predicate nodeselectioncriteria);
ActionNotification(tecton_id tecton,
                   Metadata name_val_pairs);
```

SetContactCriteria() is used to define the conditions under which "contact" with another tecton should be announced. It may only be called by the tecton's coordinator node. The syntax of the *Predicate* argument is the same as the corresponding argument to C*reate()*; but here the predicate may range over <*name*, *value*> metadata attached to tecton-spaces and/or tecton-machines and/or tectons and/or nodes.

A *ContactNotification()* upcall is delivered to a tecton's coordinator node when the *contactcriteria* predicate passed to *SetContactCriteria()* has evaluated to true. The upcall provides all the information necessary to determine appropriate action-consequent-to-contact: i.e. the tecton with which contact has been announced, the nodes in both this and the other tecton that were involved in the contact, and all the associated metadata.

The *Action()*/*ActionNotification()* APIs are used to transfer <*name*, *value*> metadata representing a "request for action" from the coordinator node to any nodes that it determines should be involved in carrying out the action. *Action()* may be called only by the coordinator node, and is usually called as a consequence of the receipt of a *ContactNotification()*. A call of *Action()* results in *ActionNotification()* upcalls being delivered to all nodes in the tecton for which the *nodeselectioncriteria* predicate is true.

4) *Example of use*

To illustrate the use of (some of) these APIs, recall the example contact-action rule in Section I: "**IF** *any of the nodes in this MANET tecton come into contact with a node of a tecton representing an 802.11 network,* **THEN** *all the MANET's nodes should reconnect to their email servers via 802.11*".

First, let us assume a supervening ontology that comprehends the concepts of "net type", "net cost", and "node capability". Assuming this, the MANET tecton's coordinator node might perform the "IF" part of the rule by calling: ***SetContactCrite*ria(t, "net_type=802.11 and net_cost=0")** (assuming the MANET tecton is identified as *t*). The coordinator then performs the "THEN" part of the rule when it receives a corresponding ***ContactNotification()*** upcall (arguments omitted here for space reasons). It might proceed by first searching for "high capability" nodes in the newly-discovered 802.11 tecton (i.e., it would scan the upcall's *involved_nodes_in_other_tect* argument for node for which "high_capability = true"). Then, it would pick one of these nodes, say *n*, and call: ***Action(t, "<change_proxy, n>", "node_id = ANY")***. This will result in all the MANET tecton's nodes receiving an ***ActionNotification(t, "<change_proxy, n>")*** upcall, which they interpret as a command (to be implemented outside the tecton framework) to reconnect to their email servers via an address to be found in metadata attached to node *n* from the 802.11 tecton.

C. Inter-tecton-machine communication

We now discuss the underlying communication patterns implied by the tecton-machine's API calls, considering both intra-tecton-space and inter-tecton-space cases.

1) Intra-tecton-space communication

At the intra-tecton-space level, the communication implied by the API calls is mainly realised by one or more gossip plug-ins. In particular, information on all the tectons hosted within a tecton-space, including their member nodes and *contactcriteria* information, is disseminated to all tecton-machines in the tecton-space. Because tecton-spaces are assumed to be limited in extent, the overheads of this are deemed acceptable.

In addition, tecton-machines collectively monitor each other to detect crashes of tecton-machines that are supporting coordinator nodes. If such a crash is detected, the remaining tecton-machines use an election protocol plug-in to elect new coordinator nodes [16]. Finally, ongoing changes to <*name*, *value*> metadata are gossiped to all tecton-machines, and *contactcriteria* predicates are evaluated lazily—i.e., when changes are received.

2) Inter-tecton-space communication

Some of the tecton-machines in each tecton-space periodically emit service discovery probes using a service discovery plug-in; correspondingly, tecton-machines listen for probes from external tecton-spaces. Whenever this results in a potential inter-tecton-space handshake, the tecton-machines involved (subject to mutual validation via a security plug-in) exchange locally-hosted *tecton_id*s and associated metadata. Updates to these are also sent on occurrence, so that timely information is available when evaluating contact predicates.

In addition, where a tecton-machine hosts a node that is a member of a "remote" tecton (i.e. a tecton whose coordinator node resides outside this tecton-machine's tecton-space), it pushes metadata updates associated with this node to the remote tecton-space. This enables *contactcriteria* predicates to be properly evaluated across all nodes regardless of where they are hosted. All these patterns of inter-tecton-space data exchange remain scalable because only "pairwise" information exchanges are involved—i.e. they don't involve transitive closures over tecton-spaces.

D. Discussion

With the "generality" property of tectons firmly in mind, the tecton-machine's runtime API is *purposely minimal*, so that it is capable of implementation in a range of environments, including tiny embedded devices. The API therefore eschews many issues of apparent significance, leaving these to the "high-levels" of the architecture (see Section II). In particular:

- *System behaviour*: The API knows nothing of the behaviour of the "real system" being represented by a tecton. For example, it has no notion of a "service interface" offered by the system. It knows only that the system spans a number of tecton-machines.
- *Node behaviour*: Similarly, the API knows nothing of the behaviour of "real-system nodes". As far as a tecton-machine is concerned, a node is simply a piece of client code registered with it in association with a *node_id*. The purpose and behaviour of any corresponding "real system node" (e.g. packet forwarding, controlling a mobile device, representing a sensor, …) is of no interest.
- *Contact*: The detection of contact is semantically empty for the tecton-machine (see example in Section III.B.4). A tecton-machine is only responsible for the distributed evaluation of predicates defined over metadata attached to the tecton-related abstractions (i.e., tecton-spaces, tecton-machines, tectons and nodes).
- *Action-consequent-to-contact*: Similarly, a tecton has no notion of what action (e.g. composition etc.) to take when contact is announced (again, see the example in Section III.B.4). Rather, it simply serves as a "signaling protocol" that delivers <*name*, *value*> metadata to the nodes deemed responsible for performing the desired action.
- *Horizontality/verticality*: The API does not even understand the concept of vertical or horizontal contact/interaction that was discussed in Section I.

The "missing behaviour" in all cases is provided by the high levels of Fig. 2. In particular, the role of ontologies is central in imposing semantics on the foundational functionality provided by the tecton-machine. We believe that the separation of concerns implicit here is crucial in the design of a programming model for distributed systems of systems.

In summary, the high levels provide: i) domain-specific programming concepts to specify tectons, and to formulate rules that determine when tectons are deemed to have made contact, and what should happen when they do; ii) associated ontologies that capture domain-relevant information concepts; iii) specification of the semantic of "contact" in terms of DSL/ontology statements; iv) local, per-node, monitoring of state relevant to contact establishment, and updating corresponding metadata when this changes; and v) provision of per-node code to respond to calls for action-consequent-to-contact.

In turn, the low levels provide: i) a generic distributed run-time representation of "systems" and their associated "nodes"; ii) detection and notification (to a coordinator node), of inter-system "contact", given that requisite metadata has been provided by a high level compiler as <*name*, *value*> metadata; iii) a guarantee that there will always be a live coordinator node in charge of the tecton; and iv) signaling to aid the distributed organisation of action consequent to contact.

IV. Related Work

We are not aware of work that is taking a closely similar approach to the programming of systems of systems. However, there is work of relevance within the middleware and the networking communities. In the middleware community, there has been substantial work on using DSLs for the specification of network overlays (e.g. [7], [8]), and even predicate-based abstractions of multi-node systems [9], [10]. However, this work tends to focus on the specification of *individual* overlays rather than on their composition. Work in [11] does consider one aspect of composition, but this is aimed at sharing modules that may contribute to multiple overlays, rather than opportunistic composition of whole systems.

Within the networking community, the ANA project [5] defines the concept of a *network compartment*: an encapsulated, composable, distributed entity, that transparently forwards packets across a network. This shares with tectons the notion of a distributed composable unit, but lacks our notion of spontaneous, opportunistic, composition via contact-action rules; and also lacks the generality we seek. In addition, the delay tolerant networking community (e.g. [6], [17]) has proposed numerous scenarios and solutions involving loose coupling of separated "systems", but tends to focus on protocol issues and to lack generalisable programmatic abstractions for opportunistic system composition.

Finally, there is work on programmatic system-of-systems composition in the formal methods community (e.g. work at Pisa [15]); but there remain numerous challenges in taking this work through to viable implementation.

V. Conclusions

We have outlined the design of a programming approach to the dynamic construction of systems of systems. We have focused in this paper on the design of a low-level runtime (the tecton-machine) that supports the wider programming system architecture of Fig. 2. The essence of our approach is to *wrap* individual "systems" (e.g. sensor networks, MANETs, network overlays) in a unified manner, and to equip the wrapping with declaratively-specified rules that capture the circumstances under which the system would benefit from interacting or composing with other systems, and the manner in which this should be done. The intent is that this approach will form the basis of an ecology of autonomous "systems" that opportunistically interact depending on what they encounter in their environment, so that fit-for-purpose systems of systems can arise spontaneously in a bottom-up manner.

We are developing the tecton concept in the context of the EU-funded "Dionasys" project, in collaboration with the Universities of Neuchatel, Bordeaux and Cluj-Napoca. The project is focusing on composable overlays both in the internet and in WSNs. The target environment for our implementation of the tecton-machine on WSN devices is our Lorien OS [12], which supports "code (re)deployment technology", as required for the full realisation of the lower level of the tecton architecture. We employ the SplayNet system [13] for the bootstrapping of tecton-machines and tecton-spaces.

Our starting point for exploring the DSL/ontology layers is provided by [14], and by the Splay language [13] which aims to ease rapid prototyping of distributed systems. We are also currently defining ontologies in the domains of Software Defined Networking (SDN) and multi-cloud environments.

References

[1] Maier, M., "Architecting Principles for System of Systems", Systems Engineering 1 (4): pp 267–284, 1998.

[2] Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D.L., Sirin, E., Srinivasan, N., "Bringing semantics to web services with OWL-S", World Wide Web Journal 10, pp 243–277, 2007.

[3] Cortes, C., Blair G., Grace, P., "A Multi-protocol Framework for Ad-hoc Service Discovery". Proc. 4th Intl. Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC '06), Melbourne, Nov 2006.

[4] Singh, S., Kurose, J.F., "Electing "good" leaders", J. of Parallel and Distributed Comp. 21, pp 184–201, 1994.

[5] Schmid, S., Schuetz, S., Zimmermann, K., Nunzi, G., Brunner, M., "Autonomic and Decentralized Management of Wireless Access Networks", IEEE Transactions on Network and Service Management, 4 (2), pp 96-106, Sept 2007.

[6] Fall, K., "A Delay-Tolerant Network Architecture for Challenged Internets", Proc. SIGCOMM, Aug 2003.

[7] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I. "Towards a Common API for Structured P2P Overlays". Proc. 2nd International Workshop on Peer-to Peer Systems (IPTPS), pp 33-44, Berkeley, CA, USA, Feb 2003.

[8] Li, B., Guo, G., Wan, M., "iOverlay: A Lightweight Middleware Infrastructure for Overlay Application Implementations", Proc. ACM/IFIP/USENIX International Middleware Conference (Middleware 2004), pp 135-154, Toronto, Canada, Oct 2004.

[9] Picco, G.P., Mottola, L., "Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks", Proc. 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS '06), San Francisco (CA, USA), Springer Lecture Notes on Computer Science vol. 4026, pp 150-167, June 2006.

[10] Pasquet, M., Maia, F., Rivière, E., Schiavoni, V., "Autonomous Multi-Dimensional Slicing for Large-Scale Distributed Systems", *Proc. DAIS'14*, Berlin, Germany, June 2014.

[11] Rivière, E., Baldoni, R., Li, H., Pereira, J., "Compositional gossip: a conceptual architecture for designing gossip-based applications", ACM SIGOPS Operating Systems Review, Special Issue on Gossip-Based Networking, Oct 2007.

[12] Porter, B., Coulson, G., "Lorien: A pure dynamic component-based Operating System for Wireless Sensor Networks", Proc. ACM MidSens, pp 7-12, Nov 2009.

[13] Schiavoni, V., Rivière, E., Felber, P., "SplayNet: Distributed User-Space Topology Emulation", Proc. Middleware, 14th ACM/IFIP/USENIX Middleware Conference, Beijing, Dec 2013.

[14] Blair, G., Bennaceur, A., Georgantas, N., Grace, P., Issarny, V., Nundloll-Ramdhany, V., Paolucci, M., "The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems", Proc. ACM/IFIP/USENIX Middleware Conference, Lisbon, pp 410-430, Springer Verlag, Dec 2011.

[15] Börger, E., Schewe, K.-D., "Concurrent abstract state machines", University of Pisa, 2014.

[16] Taïani, F, Porter, B, Coulson, G., Raynal, M., "Cliff-edge consensus: agreeing on the precipice'", Springer Lecture Notes in Computer Science, vol. 7979, pp 51-64, 2013.

[17] IEEE WoWMoM workshop series on Autonomic and Opportunistic Computing, http://aoc2014.conference.nicta.com.au.