Dynamic Deployment and Reconfiguration of Ad-Hoc Routing Protocols

Rajiv Ramdhany, Paul Grace, Geoff Coulson, David Hutchison

Computing Department,
Lancaster University,
South Drive,
Lancaster, LA1 4WA, UK
{r.ramdhany, gracep, geoff, dh}@comp.lancs.ac.uk

Abstract. The innate dynamicity and complexity of mobile ad-hoc networks (MANETs) has resulted in numerous ad-hoc routing protocols being proposed. Furthermore, numerous variants and hybrids continue to be reported in the literature. This diversity appears to be inherent to the field—it seems unlikely that there will ever be a 'one-size-fits-all' solution to the ad-hoc routing problem. However, typical deployment environments for ad-hoc routing protocols still force the choice of a single fixed protocol; and the resultant compromise can easily lead to sub-optimal performance, depending on current operating conditions. In this paper we address this problem by exploring a framework approach to the construction and deployment of ad-hoc routing protocols. Our framework supports the simultaneous deployment of multiple protocols so that MANET nodes can switch protocols to optimise to current operating conditions. The framework also supports finer-grained dynamic reconfiguration in terms of protocol variation and hybridisation. We evaluate our framework by using it to construct and (simultaneously) deploy two popular ad-hoc routing protocols (DYMO and OLSR), and also to derive fine-grained variants of these. We measure the performance and resource overhead of these implementations compared to monolithic ones, and find the comparison to be favourable to our approach.

Keywords: Ad-hoc routing, protocol frameworks.

1 Introduction

Mobile ad-hoc networks (MANETs) employ routing protocols so that out-of-range nodes can communicate with each other via intermediate nodes. Unfortunately, it is hard to design *generically-applicable* routing protocols in the MANET environment. This is for two main reasons: First, ranging from dynamic military networks consisting of hundreds of nodes to smaller-scale multi-hop extensions of WLAN technologies through low-power sensor networks, MANETs vary widely in size, node-density, node-degree, node-mobility and bandwidth-delay characteristics (network capacity). Even within each MANET environment, the network experiences *dynamic variations in network connectivity* as nodes join and leave the network. Over time, the network size, node density, mobility patterns of the nodes, the topological

rate of change, degree of the nodes and link capacities vary, making network context dynamic and ephemeral. Second, MANETs are subject to a *diverse and dynamic set of application requirements* in terms of quality of service (QoS) demands and traffic patterns (i.e. in terms of messaging, request-reply, multicast, publish-subscribe, streaming, etc.). For example, a media-streaming application will make stricter packet latency demands on the network than a messaging application. Collaborative networking applications, on the other hand, produce intermittent and bursty data traffic but yet cannot tolerate prolonged route acquisition times when communication between users switches from instant messaging to video conferencing.

In response to these two types of pressures—from both 'below' and 'above'—MANET researchers have been proposing an ever-proliferating range of routing protocols: e.g. AODV [23], DYMO [5], OLSR [8], ZRP [14], TORA [22] and GPSR [17] to name but a few. However, none of these proposals comes close to providing optimal routing under the full range of operating conditions encountered in MANET environments; and it is becoming ever clearer that the 'one-size-fits-all' ad-hoc routing protocol is an impossibility.

We therefore believe that future MANET systems will need to employ *multiple* ad-hoc routing protocols and to support switching between these as runtime conditions dictate. Our view is that this is best achieved through a *runtime framework based approach* in which different ad-hoc routing protocols can be dynamically deployed—both serially and simultaneously—depending on current operating conditions. In our view, such a framework should further employ a fine-grained compositional approach so that ad-hoc routing functionality can be built by composing fine-grained building blocks at runtime. Such an approach would support the creation of variants and hybrids of protocols at run-time so that we can adapt to changing runtime conditions in a finer-grained manner than switching protocols. Such an approach would also support the sharing of common functionality between protocols (thus reducing both development effort and resource overhead), and ease the task of deploying and porting newly-designed protocols and protocol updates.

In this paper we propose such a framework. The specific goals of the framework, which is called MANETKit, are:

1. To support the dynamic deployment of ad-hoc routing protocols, both serially and simultaneously, and also to support their fine-grained dynamic reconfiguration.
2. To do this while achieving comparable performance and resource overhead to equivalently-functioning monolithic implementations.
3. To further support protocol diversity by shortening the protocol development cycle and the time to port protocols to different operating systems.

This paper is an in-depth motivation, description and evaluation of MANETKit. The remainder of the paper is structured as follows. Section 2 makes the case for MANETKit in more detail, based on an analysis of the design space of ad-hoc routing protocols and a survey of existing protocol construction frameworks. Section 3 then provides brief background on the key technologies and concepts underpinning our framework, Section 4 presents the framework itself, and Section 5 illustrates its use by means of case study implementations of some popular ad-hoc routing protocols

(OLSR and DYMO). Section 6 then provides an empirical evaluation against the three goals specified above, and Section 7 offers our conclusions.

2 Related Work

Ad-hoc Routing Protocols. The design space of ad-hoc routing protocols can be divided into three broad categories:

- *Proactive* (or table-driven) protocols (e.g. [8]) continuously evaluate routes from each node to all other nodes reachable from that node.
- *Reactive* (or on-demand) protocols (e.g. [5]), on the other hand, discover routes to destinations only when there is an immediate need for it.
- *Hybrid* protocols (e.g. [14]) combine aspects of both proactive and reactive types—e.g. by employing proactive routing within scoped domains and reactive routing across domains.

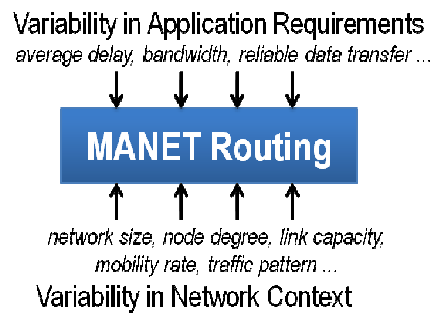


Fig. 1. Variability pressures from above and from below

As mentioned in the introduction (also, illustrated in Fig. 1), the pressures that are driving the proliferation of ad-hoc routing protocols are coming from both 'below' and 'above'. From 'below', the biggest determining factor in which protocol is the most appropriate is the size of the network: generally, proactive protocols are better suited to smaller networks, reactive ones to larger networks, and hybrid protocols to networks that can structured hierarchically. But where the network *varies* in size (e.g. grows), an initial choice of protocol (e.g. proactive) can become sub-optimal. As another example, a reactive protocol will do well where pairs of interacting source-destination nodes (i.e. an influence from 'above') tend to be stable, while proactive protocols are typically better where interaction patterns are more dynamic (although only where the network is not too big). Another pressure from 'above' is the need to integrate ad hoc routing protocols with higher level environments such as transport or middleware frameworks (cf. 'cross-layer interaction'). This approach, increasingly favoured for the design of MANET protocol stacks, departs from the 'strict protocol layer separation' advocated for wired-network protocols. Because of bandwidth limitations in MANETs, increased two-way vertical communication within the MANET protocol stack is encouraged [37-39] to remove inefficiencies and reduce the amount of horizontal communication for higher layer protocols. Thus, peer-to-peer

services running over MANETs tend to prefer proactive protocols for their extensive topological information [3]; service discovery protocols favour reactive protocols to piggyback their service discovery packets on route queries; and applications requiring QoS differentiation can benefit from intelligent path selection as enabled by multipath routing algorithms like TORA [22] or Multipath DYMO [10] —although these carry overhead that is unnecessary for other applications (or application use-cases).

As well as proposing many new protocols in each of the above categories, researchers have since investigated numerous variations on already-existing protocols. For example, path accumulation [5], pre-emptive routing [12], multi-path routing [10], power-efficient routing [33], fish-eye routing [34], and numerous styles of flooding [8, 26, 1, 15] are examples of techniques that can be 'switched on' to improve a particular property of an underlying base protocol under certain operating conditions, but which may be counter-productive under other conditions. Flooding (which is typically used to propagate control information) is a particularly rich area in this respect. For example, Multipoint Relaying [8] is good at reducing control overhead in denser networks, whereas Hazy-Sighted Link State [26] provides better performance as the network grows in diameter. Various epidemic/ gossip algorithms (e.g. [1] [15]) can also be applied in this context.

The key conclusion is that *no single protocol or class of protocols is well suited to more than a subset of the operating conditions to be found in any given MANET environment at any given time*.

Protocol Frameworks. We are not alone in recognising the benefits of the framework approach for ad-hoc routing protocols: MANET researchers have recently developed a number of such frameworks, prominent among which are ASL [18] and PICA [4]. ASL, for example, enhances underlying system services and provides MANET-specific APIs such that routing protocols can be developed in user-space. PICA alternatively provides multi-platform functionality for threading, packet queue management, socket-event notifications to waiting threads, and network device listing, as well as minimising platform-related differences in socket APIs, and kernel route table manipulation. We have therefore found these useful inspiration for the design of analogous functionality in MANETKit. In addition, the popular Unik-olsrd [32] implementation of OLSR supports a plug-in framework which has been well used by researchers [33, 34]. However, unlike MANETKit, all of these frameworks offer purely design-time and implementation-time facilities; they do not address the *run-time configuration/ reconfiguration* support which we argue is key to the support of future MANET environments.

As well as MANET-specific frameworks, a range of more general protocol composition frameworks have been proposed. These fall mainly into two lineages: the x-kernel [30] to Cactus [2] lineage, and the Ensemble [27] to Appia [24] lineage. Unfortunately, all such frameworks are of limited relevance to our ad-hoc routing domain. This is for two main reasons. Firstly, general purpose frameworks do not address the resource scarcity inherent to MANET environments. Cactus, for example, is significantly more resource hungry than MANETKit: the C version of Cactus occupies 466KB empty, whereas MANETKit supporting two ad-hoc routing protocols occupies only 236.6KB (see Section 6.2). Secondly, they focus on traditional end-to-end protocols such as TCP/IP and do not support or emphasise

routing-specific functionality such as that supported by, say, PICA (see above). In addition, they offer poor support for the fact that application execution and packet forwarding are inherently concurrent in ad-hoc routing protocol deployments: Appia supports only a single-threaded concurrency model, and Cactus, while it supports multi-threading, leaves concurrency control entirely up to the developer. Furthermore, Appia's strictly layered model is problematic in the ad-hoc routing protocol domain where cross layer optimisation is important. Finally, the Click modular router [40] is a noteworthy framework that enables a network protocol to be assembled from individual packet processing elements. The elements embody simple router functions such as packet classification, queuing, scheduling and interfacing with network interfaces. Routing protocols (including ad hoc routing protocols) are composed by connecting appropriate elements in a graph; packets flow along the edges of the graph and are the principal means of inter-element interaction. There are, however, a number of drawbacks to Click for our purposes. In particular, it is not specifically designed for the MANET domain; it is not easy to add new components at runtime; and it lacks support for multiple threads. It also lacks richness in inter-component interactions: its use of packet transfer interfaces as the main mode of interaction can lead to unnatural configurations such as performing route look route lookups by making packets flow through a "Route Table" element.

3 Background Concepts Underpinning MANETKit

Before introducing MANETKit, this Section briefly covers essential background that underpins our framework. This mainly consists of the OpenCom software component model [9] and its associated notion of 'component frameworks' which we use as the basis of modularisation, composition and dynamic reconfiguration in MANETKit. We also introduce the 'CFS pattern' [31] that we use to structure the implementation of ad-hoc routing protocols.

OpenCom and Component Frameworks. OpenCom is a run-time component model that uses a small runtime kernel to support the dynamic loading, unloading, instantiation/destruction, composition/decomposition of lightweight programming language independent software components. Components have *interfaces* and *receptacles* that describe their points of interaction with other components. OpenCom also supports so-called *reflective meta-models* to facilitate the dynamic inspection and reconfiguration of component configurations. In particular, it employs (*i*) an *interface meta-model* to provide runtime information on the interfaces and receptacles supported by a component; and (*ii*) an *architecture meta-model* that offers a generic API through which the interconnections in a composed set of components can be inspected and reconfigured. *Component frameworks* [16] (hereafter, CFs) are domain tailored composite components that accept 'plug-in' components that modify or augment the CF's behaviour. Plug-ins are inserted and manipulated by means of an 'architecture' reflective meta-model that is exported by each CF. Crucially, CFs actively maintain their integrity to avoid 'illegal' configurations of plug-ins—attempts to insert and manipulate plug-ins are policed by sets of integrity rules registered with

the CF. CFs adopt the *configurator pattern* [36] by including a *configurator* component that represents the component architecture and also acts as a unit of autonomy for making local decisions on when and how to change the framework based on a set of *policies*. As CFs are themselves components, they can easily be nested: i.e. more complex CFs can be built by composing simpler ones; and they can be loaded and unloaded dynamically so that only functionality that is actually instantiated needs to be paid for. Full detail on OpenCom and CFs is available in the literature [9].

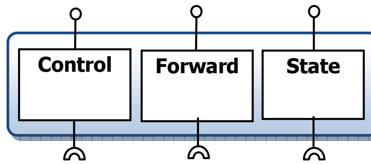


Fig. 2. MANETKit's Control-Forward-State (CFS) pattern (interfaces are shown as dots and receptacles as cups)

The Control-Forward-State Pattern. We have identified an architectural pattern called *Control-Forward-State* ('CFS' for short; see Fig. 2) that we have found useful in the structuring of protocol implementations in MANETKit. We first used the pattern in a different context in our GRIDKIT platform [31]. In the CFS pattern, the *Control* (C) element encapsulates the algorithm used to establish and maintain a virtual network topology (as often maintained by ad-hoc routing protocols); the *Forward* (F) element encapsulates a forwarding strategy over this topology; and the *State* (S) element gives access to protocol state (such as the neighbour list that embodies the virtual topology). The key benefit of the CFS pattern is that it naturally captures the typical elements of an ad-hoc routing protocol and thus allows the diversity such protocols to be treated in a consistent manner. Furthermore, when protocols are reconfigured it lets the C and F elements be replaced independently (e.g. maintaining the same overlay but changing the forwarding strategy, or vice versa). Additionally, the pattern naturally supports vertical stacking—e.g. for piggybacking data on the packets of a lower CFS element. Such stacking can be at a finer-grained level than that of entire CFS units: for example, the C element of a higher level CFS unit may use (and therefore be stacked on) the F element of a lower level unit. Finally, because a CFS instance is a composition of components, it is naturally realised as a CF and thus benefits from the above-mentioned integrity maintenance machinery that is available to all CFs.

4 The Design of MANETKit

4.1 Overview

MANETKit is a framework (an OpenCom CF) that supports the development, deployment and dynamic reconfiguration of ad-hoc routing protocols. It provides the developer with an extensible set of common ad-hoc routing protocol functionality (encapsulated in components), and tools to configure and reconfigure protocol graphs

implemented as nested CFs. It builds heavily on OpenCom's support for the dynamic reconfiguration of component topologies (i.e. the architecture reflective meta-model), and on the support for nested composition and structural integrity provided by CFs (via integrity rules). In addition, thanks to OpenCom's inherent programming language independence, MANETKit supports the development of protocols in different programming languages.

The below presentation is structured by first describing and motivating, in Section 4.2, MANETKit's main CF types and its approach to protocol composition at two granularity levels: *fine* and *coarse* . Section 4.3 then discusses further built-in CFs that provide library-like functionality for ad-hoc routing protocols, Section 4.4 focuses on the important issue of concurrency, Section 4.5 describes services to support protocol deployment and Section 4.6 discusses MANETKit's approach to dynamic reconfiguration.

4.2 Protocol Composition

Our approach to protocol composition builds directly on the CFS architectural pattern outlined in Section 3. This naturally leads to a two-level composition model. At a *fine-grained* level, a protocol composition is realised by composing elements within the CFS units using pluggable components. We propose a generic component template that can be extended by protocol developers to build protocol building blocks and embed them within MANETKit's CFS units. At a *coarse-grained* level, compositions of CFS units (i.e. protocol implementations) are used to provide multi-protocol routing configuration. We now discuss in detail the generic component template, followed by the two composition levels.

4.2.1 Generic Component Template

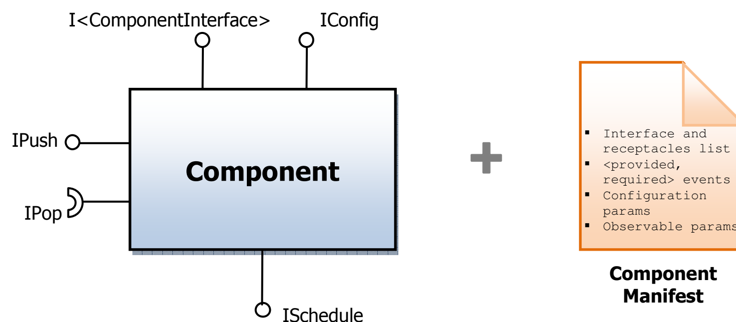


Fig. 3. Generic component template for dynamic protocol composition

The components used for composing protocols in MANETKit implement *micro-protocol modules* which are fine-grained self-contained units of protocol functionality. Micro-protocol modules are dynamically loaded, configured for use and added to a new or extant protocol composition. Only after the component bindings have been type-

checked and the integrity of the protocol configuration verified, can the component start to process protocol messages or method invocations on its interfaces. Over its lifetime, a component in a MANETKit protocol composition will additionally make method invocations on other components' interfaces, generate and/or accept event notifications (e.g. about state changes, route usage, etc.) and possibly run micro-protocol routines driven by a timer (e.g. to invalidate soft state).

To reduce inefficiencies and promote flexibility, we propose a *generic component template* (as shown in Fig 3.) that micro-protocol components in MANETKit must adhere to and extend to build the desired functionality. To enable protocol compositions to model rich component interactions whilst providing support for easy interception, re-wiring and reuse, MANETKit encourages loosely-coupled components through *event-based bindings*. To this end, as can be seen from Fig 3., each component exports a generic interface-receptacle pair, called *IPush* and *IPop* respectively, for event exchanges. Event-based bindings are implicit, asynchronous and multiparty:

- *Implicit:* To make event subscriptions implicit and automatic between components, each component specifies the type of events it requires and produces. Protocol compositions can then be automatically realised by matching the event subscriptions of its constituent components.
- *Asynchronous:* Event-publishing components need not block when generating events and event subscribers are notified asynchronously of new events.
- *Multiparty:* Unlike traditional bindings, the event-based binding allows for rich interactions whereby a group of components can subscribe to a component's set of events.

The operation of the event-based binding scheme is explained in more detail in Section 4.2.3.

The asynchronous nature of event-based bindings does not, however, suit all component interactions. Synchronous calls are useful, for example, where a component needs to perform a route lookup in a route table maintained by another component. For this reason, the generic component template also retains the synchronous mode of component interactions via standard interface bindings (shown by the *I<ComponentInterface>* interface in Fig. 3). The generic component template also exports an interface, called *ISchedule*, which allows synchronously-invoked methods to be registered with a timer service for execution at set time intervals. For instance, a component's method for emitting protocol messages or context events periodically can be automatically detected and scheduled for execution when the component is added to the protocol configuration.

Before a component can be used, it needs to be configured with the node- or protocol-specific parameters. This is achieved using the *IConfig* interface. The use of a standard interface simplifies configuration and ensures a unified configuration process across all protocols. In more detail, the IConfig interface allows *configuration strings* to be passed either at load time or at runtime to the underlying component. Configurations strings in MANETKit are comma-separated lists of arguments, where each argument is a space-separated list of objects. The objects can be keywords defining particular protocol parameters (e.g. HELLO_INTERVAL, RREQ_WAIT_TIME), strings, time/delay values, booleans (to turn on/off an optional

routing feature), IP addresses in conventional or CIDR format. Explicit error handlers are supported that are triggered if an error is encountered in parsing a configuration string.

Finally, in order to improve their reusability and ease of deployment, components make explicit the following meta-data about themselves in an associated 'Component Manifest':

- List of interfaces and receptacles supported – this enables components to be treated as black boxes, and allows their services and service requirements to be dynamically discovered. The list is only indicative though, as components can dynamically instantiate new interfaces and receptacles over their lifetime.
- Provided and required event type tuples – these are used for event-based binding as discussed above.
- Configuration parameters – these are keywords that are recognised by the configuration parser discussed above.
- Observable parameters – these embody context information that is gathered by the component; they also specify an information capture semantic (e.g. on-demand, or periodically through method invocations, or ad hoc notifications through event subscriptions).

4.2.2 Fine-grained Composition

MANETKit offers a generic sub-CF, called the 'ManetProtocol CF' based on the CFS design pattern for the fine-grained composition of ad-hoc routing protocols. Internally, the C, F and S elements of each ManetProtocol instance are structured individually in terms of composition of generic component template instances (see Fig. 3). For the C element, we provide a generic sub-CF called *ManetControl* which encapsulates a number of areas of functionality (especially event management) that are expected to be common across a range of ad-hoc routing protocols. For example, ManetControl's C component provides generic operations to initialise, start or stop a protocol's execution, maintains an Event Registry that supports the automatic event binding mechanism used for intra-ManetProtocol and inter-ManetProtocol interactions (inter-ManetProtocol interactions are described in Section 4.2.3), and offers operations to push/pop events across these bindings. The S element is fulfilled by a generic state sub-CF, called the *ManetState* CF, that can be configured to store protocol state such as routing entries, link state information and message history to name but a few. The F element which exports forwarding and data packet piggybacking services over the logical topology maintained by the ManetProtocol, is much more specific to individual protocol implementations; therefore there is less value in providing richly a configurable sub-CF in this area.

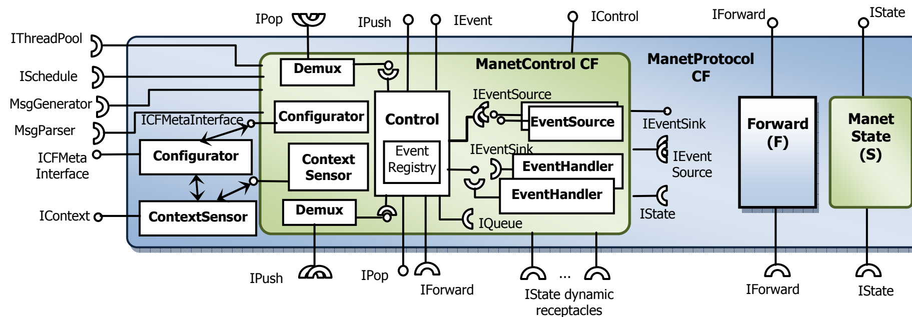


Fig. 4. Fine-grain protocol composition (i.e. within a ManetProtocol CF instance)

In general, each new ManetProtocol instance comes with default machinery and settings that can be modified or replaced depending on the developer's specific requirements. Subsequent tailoring of a new instance is easily achieved through the ManetProtocol reflective meta-interface (outer *ICFMetaInterface* in Fig. 4.) which provides facilities to inspect, configure and modify the internal CFS units of the protocol. Tailoring may take the form of relatively minor changes such as protocol parameter tuning (passing configuration strings to appropriate sub-CFs and components) or wholesale changes such as adding new components to the existing configuration of components and sub-CFs. This is a safe process because the integrity rules (architectural constraints) built into all the generic CFs ensure that attempts to compose them do not violate per-CF structural invariants: for example, ManetProtocol will reject attempts to add more than one C element. New integrity rules can be plugged in via the Configurator component. Aside from this common functionality, the core logic of a routing protocol implementation is embodied as a set of Event Source/Event Handler micro-protocol components within the ManetControl CF (Event Sources only emit events—typically driven by a timer—whereas Event Handlers process events, and may emit further events in response). These Handler components are added and configured using the reflective meta-interface of the ManetControl CF as per its integrity rules – only one instance of the ContextSensor and Control components are permitted.

In general, interaction among these fine-grained components, for the reasons specified in Section 4.2.1, follows a dual approach: individual CFS elements and sub-elements communicate either via events or via direct calls. As an illustration of this interaction duality, micro-protocol components, although themselves Event Handlers and Event Sources, access local protocol state through synchronous invocations via ManetControl's *IState* receptacle to circumvent potential delays in event queuing and processing.

4.2.3 Coarse-grained Composition

At the coarse-grained level, MANETKit uses two key (sub)CFs to compose multi-protocol 'stacks': a so-called 'System CF' that encapsulates common system-related functions; and the ManetProtocol CF (described in Section 4.2.2) that is instantiated and tailored for each ad-hoc routing protocol developed in MANETKit. As shown in Fig. 5, a MANETKit deployment running on a node typically comprises a number of composed ManetProtocol instances atop a single System instance. ManetProtocol

instances may be placed at the same level or stacked on top of each other. Communication between CFS units within a MANETKit deployment—e.g. the flow of packets or context information—is carried out using *events*[1]. The set of events supported in a given MANETKit deployment is based on an extensible polymorphic ontology. To leverage existing efforts in the direction of consolidation of ad-hoc routing protocols, we employ the increasingly-used PacketBB packet format [7] as the basis of our event structure. As illustrated in Fig. 6., events consist of a standard event header grouping common fields followed by a collection of self-describing Type-Length-Value and Address Blocks that contain more specific attributes. This ensures that event and protocol message parsing are kept generic as well.

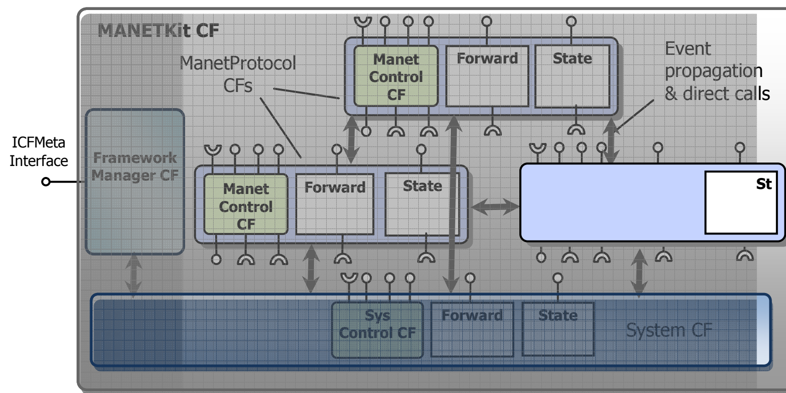


Fig. 5. Coarse-grained protocol composition in MANETKit

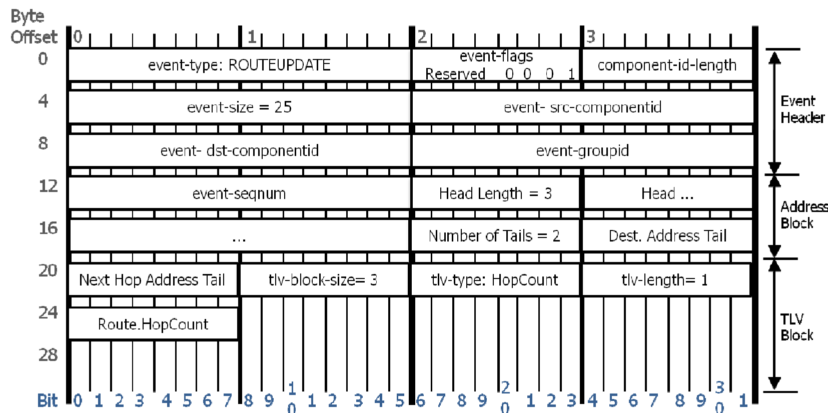


Fig 6. A ROUTE_UPDATE event based on the PacketBB [7] generalised format

---

[1] As well as using events as discussed in this section, it is possible to make direct calls from one CFS unit to another. Such calls are typically used for 'out of band' purposes such as obtaining state from another's S element. Direct calls typically benefit from OpenCom's 'interface meta-model' to dynamically discover interfaces at runtime.

Rather than being built explicitly, the organisation of a stacking topology of CFS units is derived *automatically* based on aggregated statements of the types of event provided by and required by each CFS unit. More specifically, each unit builds from its constituent elements and sub-elements a tuple: <*required-events*, *provided-events*> in which 'required-events' is the set of event types that the CF instance is interested in receiving, and 'provided-events' is the set it can generate. The required-events set of ManetProtocol instance is the union of the required-event sets of individual components composing the C, F and S elements in the CF. Similar principles apply for the provided-events set of a ManetProtocol instance. Following these rules, the event tuple of a ManetProtocol can be automatically sourced from its corresponding *IEvent* interface (See Fig. 4). On the basis of these event tuples, the Framework Manager (see Fig. 5) automatically generates and maintains an appropriate set of receptacle-to-interface bindings between protocols such that, if an event *e* is in the *provided-event* set of protocol P, and the *required-event* set of protocol Q, the Framework Manager creates an OpenCom binding between interfaces/receptacles on P and Q to enable the passage of events of type *e*[3]. Overall, the resulting loosely hierarchical organisation yields the following benefits:

- Changes in topology can be automatically updated when the event tuples on CFS units are changed at run-time (declarative automatic dynamic reconfiguration).
- The scheme naturally supports 'broadcast' event propagation (i.e. because multiple CF instances can 'require' an event of a single lower layer instance, or a lower layer instance can require an event of multiple higher-layer instances).
- It also naturally supports cross-layer interaction that omits layers, and minimises overhead where events need to pass directly between non-adjacent CF instances (avoids the need for strict layering).
- The inherent decoupling of protocols enables us to support different concurrency models without changing protocol implementations (see Section 4.4).

Finally, because it is an OpenCom CF, MANETKit can use the CF notion of integrity rules to sanity check the configuration defined by the provided-event / required-event mechanism. For example, we might use this mechanism to ensure that only one instance of a reactive routing protocol exists in a given MANETKit deployment.

4.3 Other Key Frameworks

We now briefly describe three further key CFs supported by MANETKit. These are the *ManetState CF* used in ManetProtocol instances, which manages protocol state, the above-mentioned *System CF*, a singleton CF that abstracts over low level systems oriented functionality; and the *Neighbour Detection CF*, which provides generic support for network topology management. Aside from these, MANETKit provides a

---

[3] This is a simplification. The design is slightly more complex—for example, to allow components to *exclusively* receive (require) a given event, meaning that other components would not receive the event even if it were in their required set. A mechanism to avoid loops is included for cases where a component provides and requires the same event type.

wide range of other utility components/CFs such as timers, threadpools, routing tables and queues.

The ManetState sub-CF In adopting the CFS pattern for ManetProtocol instances, we make a conscious effort to separate a protocol's state from its logic. In this way, the state is not dispersed and cloistered across a multitude of micro-protocol components, but is readily accessible within the ManetProtocol CF. However, MANET protocol state does not particularly lend itself to commonality. Ad hoc routing and supporting protocols maintain (sometimes radically) different state representations such that providing external access to it via a common standard interface is somewhat problematic. For example, apart from the routing table, OLSR maintains information about link state, multipoint relay nodes, multiple network interfaces and neighbour lists. DYMO, on the other hand, maintains information about pending route requests and blacklisted nodes in addition to routing entries. To add more complexity to state management in MANET protocols, protocol state changes usually indicate changes in the local network topology which may be of interest not only to the encapsulating ManetProtocol but to other protocols as well. We, therefore, define a generic sub-CF, called *MANETState*, that provides a unified though abstract approach to the management of state in each MANET protocol. More specifically, the MANETState CF exports a standard *IState* interface that provides methods to perform the following: *i)* create tables for storing arbitrary protocol state, *ii)* clear/drop specified tables on request, and *iii)* perform lookups and insert/delete operations on specified tables. In addition, the MANETState CF comes along with default machinery and settings to invalidate/delete table entries after set timeout values and generate events about changes in table membership if requested. The invalidation/deletion timeout periods and event types are specified at configuration time when appropriate tables are created to hold the ManetProtocol's state. Whilst various data structures (e.g. linked lists, doubly-linked lists) can be plugged in as underlying implementations of the state tables, a default setting (e.g. AVL trees) that suit lookup intensive state tables can also be specified.

The System CF. As we have seen, the System CF (see Fig. 7) is a base layer CFS unit on top of which ManetProtocol instances are stacked. Thanks to MANETKit's abstraction of inter-component communication, the System CF itself and ManetProtocol instances above it, need not be aware of the kernel-user boundary or whether the System CF itself is implemented as a kernel or a user-space module. The main role of the System CF is to facilitate portability by acting as a surrogate for OS-specific functionality such as thread management and routing environment initialisation. Its C component provides OS-independent operations to initialise the host's routing environment (e.g. IP forwarding, ICMP redirects) and provide access to system-oriented context information to inform dynamic reconfiguration (e.g. signal strength, packet loss rate). Its S component provides operations to manipulate the kernel routing table, and query/list network devices. Its F component provides send/receive primitives for the exchange of protocol messages that abstract over the use of multiple network technologies. Both the C and F elements provide and require events which higher-level ManetProtocol instances can 'specify' in their event tuples. The raising and capturing of events is ultimately grounded in mechanisms such as

network sockets, packet capture libraries (such as libpcap), and packet filters (like Netfilter in Linux or the NDIS intermediate driver in Windows).

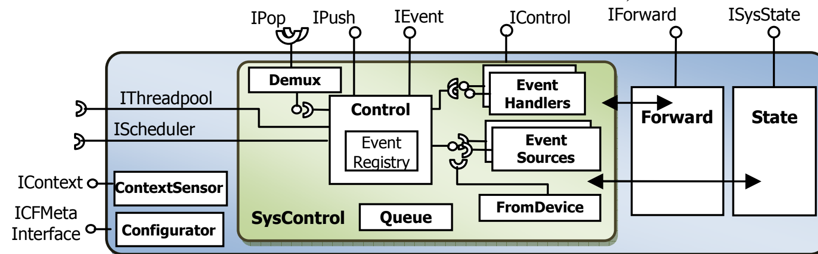


Fig. 7. The System CF

The Neighbour Detection CF. This is a generally-useful ManetProtocol instance that gathers local network topology information. More specifically, it maintains information on neighbouring nodes that are one or two hops away. Examples of this information might be the state of the links the node shares with its neighbours and their respective costs (round-trip delay, queuing delay and link capacity). If a link-state routing ManetProtocol instance is deployed, it provides the information about the links to directly-connected neighbours to other nodes for least-cost path computation. Locally, however, it uses this information to generate events to notify ManetProtocol instances about link breaks with lost neighbours for purposes of route invalidation. The information maintained by the CF is also useful as a means of optimising flooding approaches such as Multipoint Relaying. It is designed to be pluggable so that alternative mechanisms can be applied where appropriate (e.g. HELLO message based, or link layer feedback based). The CF additionally offers a useful means of disseminating information periodically to neighbours via piggybacking. For instance, AODV implementation might piggyback routing table entries so that neighbours can learn new routes. As another concrete example, under power-aware routing settings, each node can exchange battery-level information across HELLO messages.

4.4 Concurrency

MANETKit's concurrency provision is strictly orthogonal to the basic structure of the framework. This allows the use of alternative *concurrency models* within the framework, which in turn enables us easily to adapt the framework to different deployment environments. Regardless of which concurrency model is selected, the user-provided parts of a ManetProtocol instance can always be assumed to run as a single critical section. This has the beneficial effect that Event Handlers can always be assumed to run atomically.

In more detail, MANETKit supports the following concurrency models: single-threaded, thread-per-message or thread-per-ManetProtocol. Note that these designations apply only to the handling of events originating from 'below' the selected MANETKit instance (i.e. originating from the System CF): regardless of the concurrency model in use, it is always possible to use multiple threads to call

MANETKit from above. In the *single-threaded* model, all ManetProtocol instances rely on a single thread hosted by the System CF. In cases where an event needs to be passed to more than one higher-layer ManetProtocol instance, the same thread is used to call each ManetProtocol instance in turn. Besides the obvious benefit of the absence of race conditions, this model potentially allows MANETKit to be applied in primitive low-resource environments such as sensor motes.

In the *thread-per-message* model (a slight variant of this, called the *thread-per-n-messages* model, is midway between single-threaded and thread-per-message) distinct threads are used to shepherd individual events up the protocol graph. Where an event needs to be passed to more than one ManetProtocol instance in the layer above, a new thread is created for each, thus providing more concurrency than the single threaded model. Regardless, events are always processed in the same FIFO order so that ManetProtocol instances sharing the same interest in a set of events all process them in the same order.

Finally, in the *thread-per-ManetProtocol* model the ManetProtocol instance instantiates its own dedicated thread and an associated FIFO queue in which to store waiting events. A thread passing an event from a ManetProtocol instance in the layer below will immediately return, with the event being handed off to the higher-layer ManetProtocol's dedicated thread/queue. The thread-per-ManetProtocol model represents an intermediate point in terms of protocol throughput and resource overhead between the single-threaded model (low resource overhead and low protocol throughput) and the thread-per-message model (high resource overhead and high protocol throughput).

To select either of the single-threaded or thread-per-message model it is only necessary to ask the Framework Manager to use one or other model, and the selected model is applied throughout the MANETKit instance. The thread-per-ManetProtocol model, on the other hand, can be selected on a per-ManetProtocol instance basis, and will function the same regardless of whether the System CF uses one or more threads.

4.5 Dynamic Protocol Deployment

In addition to its composition, interaction and concurrency facilities, MANETKit provides a number of loadable services to enable the dynamic deployment of ad hoc routing and supporting protocols. Firstly, it provides a *DatagramServer* which listens for configuration commands emanating from a node leading a distributed configuration. Such a configuration command typically consists of a declarative statement of the new protocol stack composition, a selected concurrency model and parameters to configure each protocol being deployed. On reception of a new configuration to deploy, the Framework Manager uses the *OpenCom Kernel* service to load and instantiate sub-CFs and components for new ManetProtocol instances. If a particular micro-protocol component is not available for loading, the Framework Manager may also request it from neighbouring nodes in the form of mobile code. Inter-ManetProtocol bindings are created as per the automatic approach described in Section 4.2.3. Timer-driven methods in the new ManetProtocol instances are registered with a *TaskScheduler* service whilst their protocol message generation/parsing requirements are fulfilled by binding to a *MessageParser* service. If either of the thread-per-message and thread-per-ManetProtocol concurrency model are selected, a *Threadpool* service is loaded (if not already present) to satisfy the

concurrency requirement. Finally, after the protocol stack composition has been sanity-checked, configuration strings consisting of protocol parameters are built by the Framework Manager and passed to each new ManetProtocol instance respectively. Then only can the start operations be invoked on each ManetProtocol instance to enable it to process events and messages.

4.6 Reconfiguration management

As different protocols are optimised for different conditions and assumptions, the opportunity arises for dynamic reconfiguration to optimise running protocol deployments—e.g. to create variants of the protocols and hybrid compositions, or even to switch the routing strategy altogether. Table 1 indicates some likely possibilities for dynamic reconfiguration that we are currently exploring. The table sets out reconfiguration strategies together with some corresponding conditions that might trigger such strategies.

Table 1. Reconfiguration strategies in response to context changes

Context Change	***Current Configuration***	***New Configuration***
Drop in bandwidth	Proactive routing e.g. OLSR	- Reactive routing. If large network diameter, use relay flooding for RREQ dissemination. - Load-aware multipath routing if bandwidth drop is due to congestion
Application QoS requirements and unstable traffic patterns	Reactive routing e.g. DYMO	Proactive routing e.g. OLSR
High packet loss rate due to link breaks	Reactive routing e.g. DYMO	Path accumulation switched on and multipath routing
	Proactive routing (OLSR)	- Schedule earlier link state updates - Reactive routing at high node mobility
Flash crowd	Proactive routing (OLSR)	HSLS flooding optimisation in OLSR for less than 500 nodes; in excess of 1000 nodes then HSR hierarchical routing
	Reactive routing (AODV)	Query localisation variant of AODV
Topology change: Large network; high node density	Reactive or proactive routing	Hierarchical routing e.g. CGSR for up to 1000 nodes and HSR in excess of 1000 nodes
Topology change: Large network, dense concentration of nodes in islands	Reactive or proactive routing	Multi-scoped operation with intra-zone proactive routing and global reactive routing. Multipoint-relaying in the dense islands
High frequency of link breaks	Reactive or proactive routing	Link stability routing protocol e.g. ABR
Battery levels below threshold	Reactive or proactive routing	Battery energy efficient routing protocol or OLSR power-aware variant

To date, however, the main focus of MANETKit has been on *enabling* the dynamic reconfiguration of ad-hoc routing protocols. A fully comprehensive MANETKit-based dynamic reconfiguration solution for ad-hoc routing protocols would involve a closed-loop control system that comprises: (i) context monitoring, (ii) decision making (based, e.g., on feeding context information to event-condition-action rules), and (iii) reconfiguration enactment. MANETKit already provides the first and last of

these elements (as described next) but leaves the decision making to higher-level software. For example, a complete reconfigurable system could be built by combining MANETKit with the decision-making machinery proposed in [13].

Context Monitoring. The System CF provides a range of event types relating to context information such as link quality, signal strength, signal-to-noise ratio, available bandwidth, CPU utilisation, memory consumption and battery levels. In addition, individual ManetProtocol instances can choose to provide protocol-specific context events. For example, our DYMO implementation provides events relating to packet loss, and the number of route discoveries initiated per unit time. MANETKit also provides a 'concentrator' for context events in the Framework Manager CF (see Fig. 2). This acts as a façade for higher-level software and also hides the fact that some low level context information might be obtained by polling rather than by waiting for events.

Reconfiguration enactment. We support two complementary methods of reconfiguration enactment. The first is by updating the <*required-events*, *provided-events*> tuples of ManetProtocol instances. This enables protocol configurations to be rewired in a very straightforward, declarative, manner, although only at the coarse granularity level. The second method is more general and supports the fine granularity level: it follows the standard OpenCom approach of manipulating component compositions—i.e. by adding/removing/ replacing components and/or the bindings between them. This is carried out through standard OpenCom and CF facilities—especially the architecture reflective model outlined in Section 3. This method of reconfiguration enactment is considerably simplified by the fact that ManetProtocol instances are critical sections which only a single thread can enter at a time (see above), thus avoiding the possibility of race conditions between a reconfiguration thread and a protocol processing thread. By ensuring that any current processing of protocol events is completed before reconfiguration operations are run and further event-shepherding threads are blocked, the critical section enables the ManetProtocol instance to be in a stable state in which reconfiguration changes can be safely made. To date our experience has been that the integrity of almost all reconfiguration operations can be ensured with this critical section mechanism alone. For very complex reconfigurations (e.g. involving transactional changes across multiple ManetProtocol instances), we can fall back on OpenCom's general-purpose 'quiescence' mechanism as described in [25].

The other commonly-cited problematic issue in dynamic reconfiguration is state management. We have found that the CFS pattern is of considerable help here as it encourages designers to factor out the state from their protocol designs and put it into the generic ManetState sub-CF. Given this, if it is required to replace one ManetProtocol instance with another while maintaining state it is often enough simply to carry over a ManetState sub-CF from the old ManetProtocol instance to the new one.

5 Implementation Case Studies

To evaluate MANETKit, we have used the framework to implement a number of popular ad-hoc routing protocols. In the first instance, as a proof of concept, we used an initial Java-based implementation of MANETKit [35] to build the well-known AODV protocol. Thereafter, to investigate the feasibility of the framework in more memory-constrained devices, we developed a C version of MANETKit (based on the C version of OpenCom) and used this to implement RFC-complaint versions of the popular OLSR and DYMO protocols. In the remainder of this Section, we describe these implementations. In doing so, we illustrate how MANETKit makes it straightforward to develop and deploy ad-hoc routing protocols, and also how variants of protocols can easily to created via dynamic reconfiguration when current operating conditions call for them.

5.1 OLSR

MANETKit's OLSR implementation is built using two separate ManetProtocol instances: one for OLSR proper and the other for an underlying implementation of Multipoint Relaying (MPR) [8] that is used by OLSR. MPR is responsible for link sensing and relay selection; and maintains state in its S component to underpin these. The OLSR ManetProtocol itself uses topology information garnered by MPR and uses the latter's forwarding services to flood topology information.

We have found that MANETKit considerably simplifies the process of writing protocols such as OLSR. This is first manifested in the separation of concerns enabled by software components in general and the CFS pattern in particular. At a finer granularity than the OLSR/MPR split we have already seen, reifying protocol state into a distinct S component clarifies thinking about protocol design (as well as easing dynamic reconfiguration), and the ManetProtocol CF's plug-in Event Handlers naturally correspond to the way designers think about protocols. It is also useful to be able to call on MANETKit's range of generic tools such as routing table templates and timers (e.g. the latter are needed to drive the OLSR Event Source components that periodically diffuse link state information across the network).

Having written the elements of the protocol, installing it in a running MANETKit deployment mainly involves defining the <*required-events*, *provided-events*> event tuples of each ManetProtocol instance. The OLSR instance provides a TC_OUT event (this corresponds to an outgoing OLSR 'Topology Change' message); and it requires TC_IN, NHOOD_CHANGE (which notifies a change in the underlying network neighbourhood) and MPR_CHANGE (which notifies a change in relay selection). The latter two event types are provided by the MPR instance. The MPR instance also provides and requires, respectively, HELLO_OUT and HELLO_IN events used for neighbour detection. Finally, the MPR instance requires POWER_STATUS events. These are context events that report the node's current battery levels; they are used to dynamically determine the willingness of a node acting as a relay to forward messages on behalf of its neighbours, this 'willingness' metric being factored into the relay selection process.

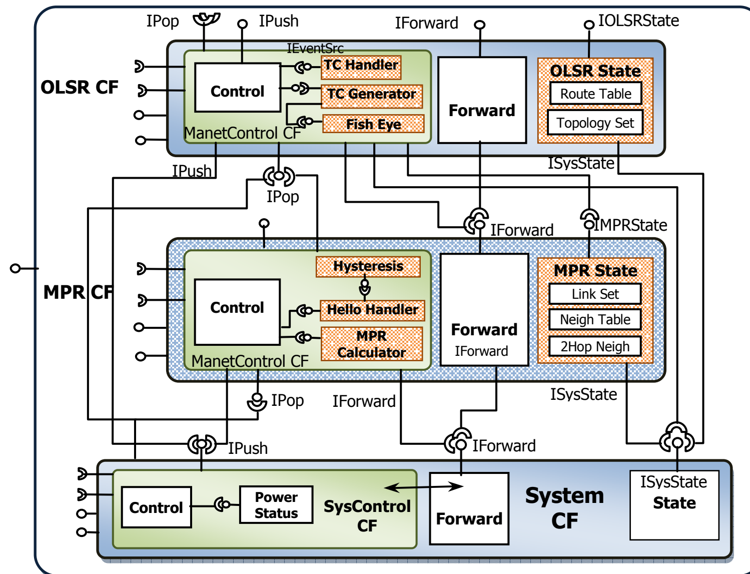


Fig. 8. The composition of OLSR in MANETKit; hatched boxes represent protocol-specific components (the rest are reusable generic components)

Protocol installation also typically entails reconfiguring some existing MANETKit CFs and if necessary, loading additional components to satisfy specific requirements. In the OLSR case, the System CF is instructed to load a 'NetworkDriver' component that requires and provides HELLO_OUT/TC_OUT and HELLO_IN/TC_IN respectively, and a 'PowerStatus' component that generates POWER_STATUS events. Fig. 8 illustrates the final protocol composition for our OLSR implementation; only the major inter-layer bindings are shown in the figure for the sake of clarity.

Protocol Variations. It is straightforward to dynamically reconfigure our OLSR implementation to better suit new operating conditions it may encounter. We describe here two such variations: power-aware routing and fish-eye routing. The *power-aware routing* variant is based on the algorithm described in [33], and aims to maximise the lifetime of a route between selected source-sink pairs within the MANET. It operates by trying to find and maintain the route between such a pair that has the least energy consumption of all possible routes. It is interesting to consider this as an OLSR variation because it is only beneficial when an application requires this particular QoS emphasis (i.e. long lifetime connectivity between particular node pairs). If there is no such requirement, or the requirement goes away because the application no longer needs it, the variation becomes a hindrance (and therefore should be removed) because it incurs significantly more overhead than standard OLSR routing. To implement and deploy the power-aware routing variation, the MPR ManetProtocol's Hello Event Handler and MPR Calculator components (see Fig. 8) are replaced by power-aware versions (the new Hello Handler determines link costs in terms of

transmission power; and this is then used by the new MPR Calculator to determine relay selection). In addition, a new 'ResidualPower' component is plugged into the OLSR CF to determine the node's residual battery level and to disseminate this to other nodes in the network via MPR's flooding service. Both adding and removing the variant behaviour is straightforward and incurs only a small number of operations on the OLSR CF's architecture reflective meta-model.

The purpose of the *fish-eye routing* variant [34] is to aid scalability when networks grow large, albeit at the cost of sub-optimal routing to distant nodes. It basically works by refreshing topology information more frequently for nearby nodes than for distant nodes. This variant is straightforwardly implemented as a component that modifies TC_OUT events according to the fish eye strategy outlined above (in fact it works by modifying the TTL and timing of OLSR Topology Change messages). The component is specified to both require and provide TC_OUT events; and so all that is required to insert it into the protocol graph is to request re-evaluation of the automatic event-tuple-based binding process. This automatically results in the component being interposed in the path of TC_OUT events passing between the OSLR and MPR CFs.

5.2 DYMO

The MANETKit configuration for DYMO consists of one new ManetProtocol instance atop the System CF. It also uses the Neighbour Detection CF that was discussed in Section 4.3. The three CF instances are configured using <*required-events*, *provided-events*> tuples is a similar manner to that already described for OLSR. For example, in order to be kept abreast of network neighbourhood changes, the DYMO instance requires a NHOOD_CHANGE event from the Neighbour Detection instance for route invalidation upon link breaks.

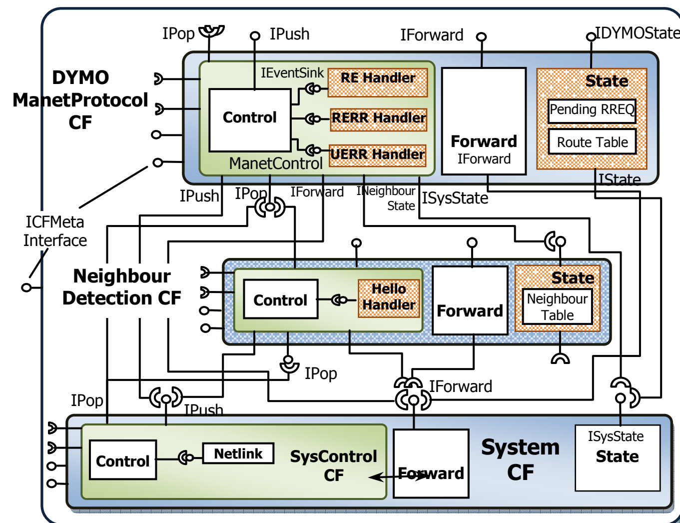


Fig. 9. The composition of DYMO in MANETKit; hatched boxes represent protocol-specific components (the rest are reusable generic components)

As a reactive protocol, DYMO requires additional machinery to ensure that route discoveries are triggered, and route lifetime updates are performed correctly. To achieve this, DYMO additionally requires the deployment of a 'NetLink' component in the System CF that is responsible for packet filtering. In implementation, this component encapsulates the loading of a kernel module that employs Linux Netfilter hooks to examine, hold, drop, etc. packets. It provides NO_ROUTE, ROUTE_UPDATE and SEND_ROUTE_ERR events which are used by the DYMO ManetProtocol instance for the purposes of (respectively): route discovery (i.e. when no route is found for an outgoing data packet), extending existing route lifetimes, and initiating route invalidations. On successful route discovery, the DYMO ManetProtocol instance sends a ROUTE_FOUND event to the Netlink component to trigger the re-injection of buffered packets into the network.

Protocol Variations. The variations we describe for DYMO are optimised flooding and multi-path DYMO. In the *optimised flooding* variant, DYMO, like OLSR, uses Multipoint Relaying as a flooding optimisation. As with OLSR, this curbs the overhead associated with broadcasting control messages when a network topology is dense, although at the expense of maintaining additional state. To apply this variation, the Neighbour Detection CF is simply replaced with the MPR ManetProtocol instance discussed in the previous Section. If a co-existing OLSR ManetProtocol instance is already deployed in the framework, then the MPR CF is directly shareable between the reactive and proactive protocols, thus leading to a leaner deployment.

The goal of the *multi-path DYMO* variant is to reduce the overhead of frequent flooding for route discovery, although at the expense of additional route discovery latency. It works by computing multiple link-disjoint paths within a single route discovery attempt, based on the algorithm described in [10]—with the notable difference that our implementation is real rather than merely simulator based. To configure multi-path DYMO, three components need be replaced (please refer to Fig. 9). Firstly, the S component is replaced with a new version that accommodates the new formats of protocol messages and routing table entries (a path list now exists for each route). Secondly, the RE (Routing Element) Event Handler is replaced with a new version that contains the logic to compute link-disjoint paths. Atomic execution of this Handler (as guaranteed by MANETKit) is essential since duplicate route requests are no longer systematically discarded but rather processed to find alternative paths. Lastly, the RERR Event Handler is replaced with a new version that handles route error events/ messages differently. For instance, on receiving a SEND_ROUTE_ERROR event, the new Handler only sends a route error message when an alternative path is not available; otherwise, it installs the new path in the OS's kernel routing table.

6 Evaluation

Section 5 has illustrated the feasibility of supporting the dynamic deployment of multiple ad-hoc routing protocols in MANETKit, and also of supporting their fine-grained dynamic reconfiguration—i.e. the satisfaction of the first of the three goals set out in the introduction has already been demonstrated. In this Section, we evaluate the remaining two goals: i.e. Goal 2: to compare favourably with equivalent monolithic implementations of ad-hoc routing protocols in terms of *performance* (Section 6.1) and *resource overhead* (Section 6.2); and Goal 3: to shorten the protocol development cycle and time to port protocols (Section 6.3).

All measurements in this Section are based our C/Linux implementation of MANETKit and use the OLSR and DYMO implementations described above. These were deployed on a testbed consisted of an 802.11b/g ad-hoc network of 5 nodes (3.2 GHz CPU with 2 GB of RAM) running Ubuntu 7.10, with an Ethernet backplane for testbed management. The 5 nodes are arranged in a linear topology: we used a combination of MAC-level filtering and the MobiEmu emulator [28] to emulate the required multi-hop connectivity. We used *Unik-olsrd* [32] as a comparator for our OLSR implementation, and *DYMOUM v0.3* [29] for our DYMO implementation. These were chosen because they are the two most popular public domain implementations of these protocols. For comparability we configured our MANETKit implementations with the single threaded concurrency model and with identical configuration parameters to the comparator implementations (e.g. identical HELLO and Topology Change intervals, and route hold times).

6.1 Performance

Our metrics for performance are (i) *Time to Process Message*—i.e. the time taken to process a protocol message from receipt to completion within an MANETKit deployment (for OLSR this is a Topology Change message; and for DYMO it is a RREQ message); and (ii) *Route Establishment Delay*—i.e. the time taken to establish a route in our testbed environment (for OLSR this is the time taken for a newly-arrived node arriving at one end of the existing linear network topology to compute a fully-populated routing table; and for DYMO it is the time taken to perform a route discovery operation under similar circumstances). The former metric is a 'micro' level indicator of the overhead of MANETKit's componentisation of the protocol processing path, while the latter is a 'macro' measure of control plane performance.

Table 2. Comparative Performance of MANETKit Protocols

	Unik-olrsd	MKit-OLSR	*DYMOUM-0.3*	MKit-DYMO
Time to Process Message (ms)	*0.045*	0.096	*0.135*	0.122
Route Establishment Delay (ms)	*995*	1026	*37*	27.3

Referring to Table 2, we can see that on the *Time to Process Message* metric, the measurements are very small in absolute terms and, as such, probably insignificant in practice. The *Route Establishment Delay* metric puts them in perspective, and shows that comparable real-world performance levels are attained by the MANETKit

implementations: MANETKit-OLSR is 3% slower than Unik-olrsd in establishing a route in our experimental set-up, whereas MANETKit-DYMO is actually 35% faster than DYMOUM-0.3. (Overall, our implementation of OLSR is slower on both metrics than the comparator, but our implementation of DYMO is faster on both.) We can conclude that that MANETKit achieves broadly comparable performance to typical monolithic implementations.

6.2 Resource Overhead

To assess the relative resource overhead of the MANETKit-implemented protocols we again compared these implementations with their monolithic counterparts—this time in terms of the memory footprints incurred. Memory footprint is the most direct measure of MANETKit's applicability for resource-constrained mobile nodes.

As can be deduced from in Table 3, MANETKit-OLSR incurs an 31% memory overhead over its monolithic competitor, and MANETKit-DYMO incurs an 48% overhead. These overheads are not surprising and are mainly due, of course, to the (necessary) inclusion of the generic MANETKit machinery and the OpenCom runtime (the latter occupies 22KB)[4]. However, as soon as we accept the premise that it is important to be able to deploy multiple ad-hoc routing protocols, as argued in this paper, we can see the benefits of MANETKit: the footprint of deploying the two protocols together in MANETKit is 8% *smaller* than the sum of the two monolithic protocol implementations; and the difference will clearly become more significant still as more protocols (plus variants) are added and the fixed MANETKit/ OpenCom overheads are further amortised. The key conclusion is that the overhead/flexibility tradeoff is already in MANETKit's favour with only two protocols deployed.

Table 3. Comparative Resource Overhead of MANETKit Protocols

	Unik-olsrd	**MKit-OLSR**	***DYMOUM-0.3***	**MKit-DYMO**	***Unik-olsrd + DYMOUM-0.3***	**MKit OLSR+ MKit-DYMO**
Memory Footprint (KB)	*136.3*	179.0	*120.4*	178.1	*256.7*	236.6

6.3 Time Taken to Develop and Port Protocols

We now evaluate the extent to which the MANETKit approach can minimise the time needed to develop and port protocols. We do this in an indirect manner—specifically, by measuring the degree of code reuse achieved across the MANETKit implementations of OLSR and DYMO.

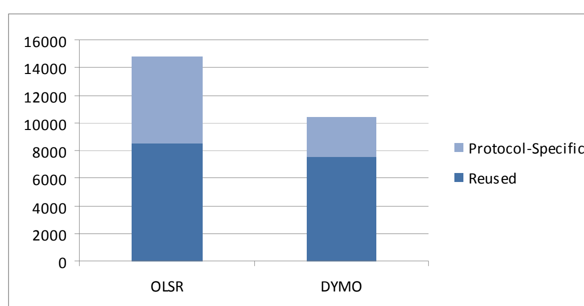
[4] Once a desired configuration has been achieved (which possibly includes multiple protocols) it is possible to unload the OpenCom kernel to free up memory space. The overheads would drop in such a case to 15% for OLSR and 30% for DYMO.

Table 4. Reused generic components in MANET protocol compositions

	Lines of Code	OLSR	DYMO
System CF Forward	1276	X	X
System CF State	702	X	X
Netlink (+ Kernel Module)	734		X
Queue	60	X	X
Threadpool	591	X	X
Timer	228	X	X
PacketGenerator	950	X	X
PacketParser	795	X	X
RouteTable	1046	X	X
ManetControl CF	827	X	X
NeighbourDetection CF	1684		X
MPRCalculator	745	X	
MPRState	3876[5]	X	
Configurator	405	X	X
Reused Generic Components	**-**	**12**	**12**
Protocol-specific Components	**-**	**4**	**5**

Table 4 gives a coarse-grained indication of the degree of code reuse by listing the generic components used in the implementation of these protocols (we also show the size of each component in terms of lines of code). In both cases, the generic components outnumber the specific ones (shown at the bottom of Table 3) by a factor of at least 2. This is especially significant because OLSR and DYMO are considered to be very different protocols.

Fig. 10 takes a finer-grained perspective by showing the number of lines of code in the generic, as well as the protocol-specific, components used by each protocol. The proportion contributed by the reusable components to each protocol's codebase is 57% for OLSR and 66% for DYMO, indicating a substantial saving in developer effort. Overall we can see that the structure of MANETKit fosters a significant degree of code reuse across protocols. Based on these measures and our knowledge of other ad hoc routing protocols we fully expect to see similar levels of reuse when we add further protocols to the framework.



[5] The reason that this component is so large is that there are several different types of table involved for the various types of data stored. There remains significant scope for optimising this figure by coalescing table handling routines.

Fig. 10. The proportion of reusable code in each protocol

7 Conclusions and Future Work

This paper has proposed a run-time component framework for the implementation, deployment and dynamic reconfiguration of ad-hoc routing protocols. It is motivated by the fact that the range of operating conditions under which ad-hoc routing protocols must operate is so diverse and dynamic that it is infeasible for a single protocol to be optimal under all such conditions. MANETKit therefore supports the serial and simultaneous deployment of multiple protocols, plus the generation of protocol variants and hybrids via fine-grained dynamic reconfiguration. It uses the 'CFS' pattern and <*required-events*, *provided-events*> tuples to allow protocols to be easily stacked or composed in a variety of ways and to be straightforwardly dynamically reconfigured. Another novel feature of MANETKit is its use of pluggable concurrency models, which enables it to be used in a variety of deployment environments with varying performance/resource trade-offs. MANETKit also helps protocol developers in the traditional way by providing a rich set of tools specifically tailored to the ad-hoc routing environment, and by isolating developers from OS specificities (including whether protocols are implemented in kernel or user space). And it also enables researchers to experiment with protocol optimisation techniques.

We have evaluated MANETKit by showing how it can be used to straightforwardly build and dynamically deploy two major ad-hoc routing protocols (i.e. OLSR and DYMO) and how these deployments can be variegated in a number of ways to suit different operating conditions. Furthermore, our empirical evaluation shows that MANETKit meets our stated goals by achieving comparable performance to monolithic implementations of the same protocols, achieving smaller resource overheads when more than one protocol is implemented in comparison to the monolithic approach, and also achieving significant code reuse across protocols (the latter being a strong indicator that the MANETKit approach should generally shorten protocol development and porting time).

In the future, our immediate plans are to integrate MANETKit into a wider dynamic reconfiguration environment by incorporating policy-driven decision making. This will be based on existing work [13], and will also include coordinated distributed dynamic reconfiguration as well as merely per-node reconfiguration. We also plan to further explore reconfiguration strategies in real-world application scenarios, to further investigate the hybridisation of protocols, and to generally gain more experience of implementing protocols in the MANETKit environment.

A version of the MANETKit software is available for download from http://www.comp.lancs.ac.uk/~ramdhany/manetkit/.

References

1. Bani-Yassein M., Ould-Khaoua M., Applications of probabilistic flooding in MANETs, International Journal of Ubiquitous Computing and Communication, Jan 07
2. Bhatti N.T., Schlichting R.D., A system for constructing configurable high-level protocols. SIGCOMM Comput. Commun. Rev. 25, 4, Oct 1995
3. Borgia E., Conti M., Delmastro F., Experimental comparison of routing and middleware solutions for mobile ad-hoc networks: legacy vs cross-layer approach, E-WIND 05
4. Calafate C.M.T., Manzoni P., A multi-platform programming interface for protocol development, 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 03
5. Chakeres I., Perkins C., Dynamic MANET on-demand (DYMO) routing, draft-ietf-manet-dymo-11, IETF's MANET WG, Nov 2007
6. Chiang C., Routing in clustered multihop, mobile wireless networks with fading channel, IEEE SICON'97, Oct 1997
7. Clausen T., Dearlove C. Jacquet P., Generalized MANET message format, draft-ietf-manet-packetbb-07 internet draft, 2007
8. Clausen T., Dearlove C., Optimized link state routing protocol, v2, draft-ietf-manet-olsrv2-03.txt
9. Coulson G., Blair G., Grace P., Taiani F., Joolia A., Lee K., Ueyama J., Sivaharan T., A generic component model for building systems software. ACM Trans. Comput. Syst. 26, 1 Feb 08
10. Galvez J.J., Ruiz P.M., Design and performance evaluation of multipath extensions for the DYMO protocol. 32nd IEEE Conference on Local Computer Networks, Oct 15, 2007
11. Garlan D., Monroe R., Wile D., Acme: an architecture description interchange language. Conference of the Centre For Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, Nov 1997
12. Goff T., Abu-Ghazaleh, N. B., Phatak, D.S., Kahvecioglu, R., Preemptive routing in ad-hoc networks, MobiCom 01
13. Grace P., Coulson G., Blair G. S., Porter B., A distributed architecture meta-model for self-managed middleware. ARM 06
14. Haas Z.J., Pearlman M.R., Samar P., The zone routing protocol (ZRP) for ad-hoc networks, Internet Draft, draft-ietf-manet-zone-zrp-04.txt, July 02
15. Haas Z.J., Halpern J.Y., Li Li, Gossip-based ad-hoc routing, INFOCOM 02
16. Joolia A., Batista T., Coulson G., Gomes A.T., Mapping ADL specifications to a reconfigurable runtime component platform, WICSA 05
17. Karp, B. and Kung, H.T., Greedy perimeter stateless routing for wireless networks, Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)
18. Kawadia V., Zhang Y., Gupta B., System services for ad-hoc routing: architecture, implementation and experiences, MobiSys 03
19. Kon F., "Automatic configuration of component-based distributed systems". PhD Thesis. University of Illinois at Urbana-Champaign, May 2000
20. Kuladinithi K., University of Bremen Java-AODV implementation, http://www.aodv.org
21. Marina M.K., Das S.R.: On-demand multipath distance vector routing in ad-hoc networks. Proc. International Conference for Network Procotols, 01
22. Park V.D. Corson M.S., A highly adaptive distributed routing algorithm for mobile wireless networks. INFOCOM 1997
23. Perkins C., Royer E., Ad-hoc On demand distance vector routing, Internet Draft rfc3561, 03
24. Pinto A., Appia: A flexible protocol kernel supporting multiple coordinated channels. ICDCS. IEEE 2001

25. Pissias P., Coulson G., Framework for quiescence management in support of reconfigurable multi-threaded component-based systems, Software, IET, vol.2, no.4, pp.348-361, Aug 08
26. Santiváñez C.A., Ramanathan R., Stavrakakis I., Making link-state routing scale for ad-hoc networks. Proc. 2nd ACM international Symposium on Mobile Ad-Hoc Networking, Oct 01
27. van Renesse R., Birman K., Hayden M., Vaysburd A., Karr D., Building adaptive systems using Ensemble. Technical Report. UMI Order Number: TR97-1638., Cornell University, 1997
28. Zhang Y., An integrated environment for testing mobile ad-doc networks. MobiHoc 02
29. Implementation of the dymo routing protocol dymoum-0.3. http://masimum.inf.um.es/?Software:DYMOUM
30. Hutchinson, N. C. and Peterson, L. L. 1991. The X-Kernel: An Architecture for Implementing Network Protocols. IEEE Trans. Softw. Eng. 17, 1 (Jan 1991)
31. Grace, P., Coulson, G., Blair, G., Mathy, L., Yeung, W.K., Cai, W, Duce, D., Cooper, C., "GRIDKIT: Pluggable Overlay Networks for Grid Computing", Proc. Distributed Objects and Applications (DOA 04)
32. Implementation of the OLSR routing protocol, Unik-olsrd website: http://www.olsr.org/
33. Mahfoudh S., Minet P., An energy efficient routing based on OLSR in wireless ad hoc and sensor networks. Proc. 22nd International Conference on Advanced Information Networking and Applications – Workshops, 08
34. Gerla,. M., Hong, X., Pei, G., Fisheye State Routing Protocol (FSR) for Ad Hoc Networks. IETF MANET Working Group Internet Draft, 02.
35. Ramdhany, R., Coulson, G., ManetKit: A Framework for MANET Routing Protocols. Proc. 5th Workshop on Wireless Ad hoc and Sensor Networks (WWASN2008), workshop attached to the International Conference on Distributed Computing Systems (ICDCS), Beijing, China, June 08.
36. Kon F., "Automatic configuration of component-based distributed systems". PhD Thesis. University of Illinois at Urbana-Champaign, May 2000.
37. Qin L. and Kunz T., "Survey on Mobile Ad Hoc Network Routing Protocols and Cross-Layer Design, " *in Technical Report of Carleton University*, Aug. 2004.
38. Borgia E., Conti M., Delmastro F., and Pelusi L., Lessons from an Ad-Hoc Network Test-Bed: Middleware and Routing Issues. In Ad Hoc & Sensor Wireless Networks, An International Journal, Vol.1, Numbers 1-2, 2005.
39. Chen, K., Shah, S. H., and Nahrstedt, K. 2002. Cross-Layer Design for Data Accessibility in Mobile Ad Hoc Networks.*Wirel. Pers. Commun.* 21, 1 (Apr. 2002), 49-76.
40. Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. 1999. The Click modular router. *SIGOPS Oper. Syst. Rev.* 33, 5 (Dec. 1999), 217-231.