A Composition Methodology for Designing Proactive Distributed Protocols

Nathanael Thompson and Indranil Gupta
Department of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL 61801
Email: nathomps,indy@cs.uiuc.edu

Kenneth Birman
Department of Computer Science
Cornell University
Ithaca, NY 14853
ken@cs.cornell.edu

Abstract

The development of methodologies for the design of distributed protocols is essential to reducing the complexity and difficulty of creating distributed systems, as well as for maturing the field. This paper presents a composition methodology for designing a range of proactive distributed protocols, with applicability to large-scale distributed systems in both the Internet (e.g., peer to peer systems) and wireless sensor networks. The methodology consists of several basic building blocks and different composition techniques which can be used to combine the blocks into solutions for distributed computing problems. The compositions preserve properties of the original components (reliability, scalability, liveness). Next, we describe a simple specification language called “Proactive Protocol Composition Language (PPCL)” which can be used to specify and compose existing source (C language) code of building block implementations in order to automate the above design methodology. We discuss how the methodology and the toolkit generated by PPCL are (i) retroactive, i.e., can generate protocols already existing in literature, and (ii) progressive, i.e., can generate new protocols. For an important distributed computing problem (membership management in large groups), we give experimental data to compare (i) a generated protocol with a hand-written protocol, and (ii) an existing protocol with an augmentation realized through PPCL.

1 Introduction

The growth and success of networked technologies such as the Internet continue to depend on efficient scalable and reliable distributed protocols running within. These protocols range from distributed middleware to network routing. While the properties and behavior of many distributed protocols is well understood, the actual act of designing a protocol remains a challenge, and is still considered by many to be an art. As of today, there are very few practical frameworks for systematically designing, and generating code for, distributed protocols.

Over the past few years, we have designed several probabilistic protocols for large-scale distributed systems, e.g., [5, 8, 9, 10, 11, 12]. Recently however, we have discovered a surprisingly simple, clean, yet powerful, composition methodology that underlies the design of an entire set of proactive protocols, including many of the protocols cited above. In this paper, we first describe this design methodology. This description is informal, owing to the design-pattern like nature of the methodology.

Secondly, we detail how a designer can use this methodology on existing source code in order to automatically generate code for new protocols. Towards this end, we present a new high level language called PPCL. The use of PPCL is not in writing the actual code for the components (which can be implemented independently in a C-like language), but instead to specify the components and compositions. It enables a designer to use the composition methodology on existing C code to automatically generate deployable code for new protocols. In effect, PPCL is an actual implementation of the above methodology (only). Finally, we present experimental results that study how the compositions preserve scalability and reliability of components, without too much of an increase in code complexity.

Several fields of engineering have attained maturity through the development of design methodologies. This includes code synthesis engines for VLSI development [1], design patterns for software engineering [7], etc. We believe that similar approaches may be needed to ensure the longevity of distributed protocols designed by the research community.

The composition methodology presented in this paper originated from a retrospective examination of probabilistic protocols that we developed for large-scale distributed systems [5, 8, 9, 10, 11, 12]. These protocols (which included leader election, multicast, aggregation, resource discovery, membership, etc.) can be generated using seven basic build-
ing blocks, each of which is either a protocol or a strategy, and by using three simple composition rules. In this paper, we describe an extended methodology with a larger number of building blocks and composition rules, powerful enough to generate protocols for Internet-based distributed systems as well as wireless ad-hoc/sensor networks. We also argue why the compositions preserve the scalability and reliability properties of the original components.

The paper is organized as follows. The next section of the paper summarizes existing composition efforts. A retrospective case study is covered in Section 3 to motivate the methodology. Section 4 explains the details of the methodology. In Section 5 discusses how properties are inherited through composition. The Proactive Protocol Composition Language is presented in Section 6, and further examples of the use of the extended methodology are in Section 7. Finally the implementation is evaluated in Section 8.

2 Related Work

A notion of composition of probabilistic I/O automata for distributed systems has been studied in the past by Lynch [20] and Wu et al [28]. The authors defined I/O automata with probabilistic state transitions and specified how these automata can be composed. In comparison, our study of probabilistic protocol composition is geared towards designing protocols for large-scale distributed systems, where the scalability and reliability are primary concerns.

Stack-based communication architectures such as the Horus [26] and Ensemble systems [19] are designed to allow network protocol layers to be stacked, with the glue between layers being a standard function call interface set. This helps to provide different notions of reliability, message ordering, etc., to the end application. Our protocol composition methodology is aimed at enhancing scalability and reliability properties of distributed protocols rather than merely providing richer properties to the application.

X-kernel [16] and Cactus [27] allow composition of network protocols in the kernel level. In x-kernel, protocol blocks interact with one another via message passing to create new protocols which extend the functionality of the kernel. Each protocol piece exports a set of function calls as a standard interface. Configurations of the kernel are specified using a textual specification language or a graph. The Cactus system allows more fine grained composition blocks than x-kernel. We target the set of protocols that exist above the kernel, typically in the middleware. Our compositions are done without interface composition in order to allow more flexibility.

[23] presents a methodology for combing proactive and reactive protocols for routing in ad-hoc wireless networks. Their methodology does not attempt to address protocol design in general, but rather focuses on the specific scenario of routing in ad-hoc networks. Their framework is a good example of one of our compositions, pairing.

Several toolkits are available for component-based software development and optimization, e.g., Knit [24]. Components have been used for designing software for routers (Click system) and operating systems (e.g., Scout). Object-oriented programming languages such as C++ and Java allow component-based program development. Software components can also be glued together through higher-level scripting languages (e.g., Python, VB, Perl) [25] to filter data streams among different components. In object request brokerage (ORB) systems (e.g., COM, CORBA, .Net), each component is an application connected to a communication network. These are all examples of software composition and do not address the issue of protocol design.

3 Case Study: A Protocol for Decentralized Membership Maintenance

In order to motivate discussion about our design methodology we present a case study that retrospectively looks at the design of the SWIM protocol for weakly consistent and scalable membership maintenance [5],[10]. We chose this protocol because it solves a well-known distributed computing problem and provides a simple example of the composition techniques.

Various distributed algorithms require each member of the group to maintain a local list of other members in the group. We call this a membership list (also a view). In a dynamic group with members constantly joining, leaving and failing silently (crash-stop), a membership maintenance protocol keeps these lists up to date. A membership protocol has two main components: (a) a failure detector protocol, and (b) a protocol for membership update dissemination. In order to generate a membership protocol, any implementations for the respective components can be fit into a template, as shown in Figure 1. We begin our discussion by first describing the building blocks for each of (a) and (b), and then showing how the building blocks are composed.

3.1 Building Block: Distributed Ping Protocol

A failure detector protocol runs constantly at each non-faulty member to detect the failures of other members. It is required to satisfy a liveness property called Eventual Strong Completeness (Completeness), i.e., crash-failure of any group member is detected by all non-faulty members that knew of the member.

The probabilistic failure detector used in SWIM is called Distributed Ping [10]. In distributed ping each member periodically attempts to ping one other member chosen uniformly at random. Target nodes are pinged directly first,
The protocol can be shown to have the following liveness properties:

- **Liveness**: (Eventual Dissemination) If the membership graph among non-faulty members stays connected after the multicast dissemination is initiated, and the sender stays non-faulty, then the multicast is eventually disseminated to all members that stay non-faulty.

- **Analytical**: [3] shows that within a number of rounds that is \(O(\log_b(N))\), each member receives the multicast with the probability of \(1 - \frac{1}{N} \cdot (1 + o(1))\) [2, 6].

### 3.3 Composition: SWIM Protocol for Weakly-Consistent Membership Maintenance

Figure 1 shows conceptually how the above described Uniform Epidemic and Distributed Ping protocols can be used together to create a membership protocol. We call such a diagram a *template*. The template describes how components are modularly combined. Notice that any combination of implementations of a failure detector and a multicast protocol can be fit into this template.

When the Distributed Ping and Uniform Epidemic protocols are fit into the failure detector and update dissemination components of Figure 1, the resultant protocol (when optimized) has the make-up of the SWIM algorithm. The optimization in this example is to piggyback outgoing messages from one protocol on messages from the other to reduce total message overhead.

From [5, 10] SWIM has the correctness properties of Eventual Strong Completeness (if a member \(M_i\) fails, each \(M_j\) that has \(M_i\) in its membership list eventually detects the failure) and Eventual Dissemination of Updates. We note that liveness and analytical properties are inherited from the base protocols that make up the SWIM protocol. In the same way the composition has analytical properties that are a concatenation of the analytical properties of the Distributed Ping and the Uniform Epidemic described in Sections 3.1 and 3.2.

### 4 Composition Methodology

The previous section showed two building blocks, Uniform Epidemic and Distributed Ping, and also introduced the Template composition. This section describes the overall protocol design methodology. The methodology provides the designer with a collection of (a) building blocks - eight categories of probabilistic/proactive protocols and strategies with well-studied liveness, scalability and reliability properties - and (b) several composition techniques that generate protocols with richer semantics, by preserving scale, reliability and liveness properties of the components. Figure 2 shows the blocks and compositions. Compositions can also be applied repeatedly to form hierarchies of protocols. A demonstration of this based on the SWIM example is shown in Figure 3.
4.2 Composition Techniques

A pair of probabilistic protocols can be composed using one of the following composition techniques. These are not formally defined, but are presented as design guidelines. The guidelines also hint at which pairs of protocols can be composed. In a later section (6) we will give one formal interpretation of the guidelines and show how it can be used to automate protocol generation.

- **Template Composition:** A “template” for a protocol can be generated from a problem specification (e.g., the group membership template in Figure 1) or from the informal specifications of two protocols one wishes to combine to achieve a protocol with collective properties. A template will typically specify the minimal interface/function calls to be exported by each component. When appropriate protocols are plugged into the template (followed by necessary optimizations), the final protocol is generated. An example w.r.t. the group membership protocol was discussed in...
Section 3.

- **Augmentation (Either Fashioning or Constraining):** A base composition \( C_1 \) could be augmented with another composition \( C_2 \) to derive a modified protocol that solves the same problem specification as \( C_1 \). Yet, the composition imparts to the (augmented) \( C_1 \) certain additional properties.

There are two types of augmentation - fashioning and constraining. For example, a Uniform Epidemic protocol can be augmented with a component that selects epidemic targets from a membership list according to (a) a probability distribution function, e.g., weighting selections based on round-trip-time estimates (fashioning) or (b) a set of constraints, e.g., eliminating certain types of members from ever being selected as targets (constraining).

For this example, constraining can indeed be seen as a special case of fashioning where certain members are assigned a round trip time of infinity to prevent them from being selected as targets. The distinction is made from a designer’s point of view.

- **Pairing:** A proactive protocol can be paired with a reactive protocol to extend the former’s behavior. For each period of the proactive protocol the reactive protocol is initiated. The resulting protocol \( C' \) has both the properties of \( C_1 \) and \( C_2 \) since the two protocols are running side by side. Optimizations can be made by piggybacking reactive messages (when available) on periodic proactive messages.

## 5 Inheriting Protocol Properties

We discuss our observations about the preservation of protocol properties under (and in spite of) composition. This discussion applies only to the compositions we have studied or mentioned in this paper, although general applicability is a possibility\(^1\). We differentiate between the **correctness** and **analytical** properties of protocol building blocks. Correctness properties include liveness properties such as eventual guarantees, e.g., those regarding detection of failures or eventual dissemination of a multicast when the view graph is connected. Analytical properties include performance metrics such as latency, probabilistic reliability, and per member message overhead.

**Property Preservation via Composition - Correctness Properties** When two building blocks are combined using the template, pairing or the augmentation techniques, correctness properties are inherited by the composition.

**Property Preservation via Composition - Analytical Properties** When two building blocks are combined using the template or pairing technique, analytical properties are inherited by the composition.

The template technique fits two components into a protocol framework (template). Similarly the pairing technique combines a proactive and a reactive protocol to execute simultaneously. In both cases the resultant composition can then be optimized to reduce message complexity etc. (e.g., the SWIM protocol). However, the use of the protocol template or pairing implies that the resultant composition is equivalent to the constituent protocols running side-by-side in the system. As such, their original properties (both correctness and analytical) are carried over to the composition. For example, the SWIM membership protocol inherits the properties from the Distributed Ping protocol that failures are detected eventually and within an average of \( \frac{1}{1-e^{-t}} \) protocol periods.

Augmentation does not affect the inheritance of correctness properties of a component \( C_1 \) as long as the composed protocol satisfies the set of preconditions specified in the original correctness property. For example, a correctness property for a uniform epidemic-based reliable multicast component says “a multicast is eventually disseminated to all members in a group if the view graph in the group stays connected”. An augmentation of the uniform epidemic with a topology aware component (e.g., where a probability distribution based on round trip time estimates is used for gossip target selection) also satisfies the above correctness property as long as the probability distribution function keeps the view graph connected.

Although the analytical properties of an augmented composition may be different from those of its components, we observe that the scale and reliability properties are similar to those of the components, in the sense that per-member overheads that vary polylogarithmically with group size suffice to achieve very high probabilistic reliability. An example is the augmented epidemic protocol discussed in Section 4.2.

## 6 Methodology Implementation

We now describe how a simple and clean high-level language called PPCL (Proactive Protocol Composition Lan-
guage) that can be used to realize the above described methodology. PPCL allows a designer to use existing C source code (for the components) in order to automatically generate code for new protocols. PPCL is in effect only used to specify components and compositions. We have implemented a compiler that takes as input source code for the individual components and a PPCL specification, and outputs code for the new protocol. Filler code is generated by the compiler itself.

6.1 Model

In order to specifically implement an otherwise informal methodology, we need to make a few assumptions about the structure of the source code. First, for proactive protocols, we assume that there exists a primary function that runs the protocol in a loop. For reactive protocols, we need an access point that can be used to hook it into other proactive protocols. We expand on these details where required.

6.2 PPCL

Although the Proactive Protocol Composition Language is simple, it is not a toy language. PPCL is powerful enough to completely specify a protocol based on the methodology outlined in this paper. The grammar of the language is in Figure 4. The following are some of the key features:

- familiar and easy to understand syntax
- supports components as base types
- supports compositions as base types
- constructs for integrating user specified filler code

Describing a protocol using PPCL can be done quickly. First the designer specifies the basic building blocks needed in the design. This is done with the component type. Next the designer specifies the compositions to be performed on the components. There is a keyword and type for each of the supported compositions. Finally the specification is given to the compiler which parses the specification and executes the compositions. It is important to note that the specification can be written at any time during the development process, before, during or after the source code has been implemented. An example PPCL file which was used in our later experiments is shown in Figure 5.

In the next sections we briefly discuss the implementation details of PPCL.

6.2.1 Components

Components are base types in PPCL. A component is specified by the source file where it is implemented and the function or set of functions that represent the component. We allow components to be specified by a set of functions because components may have different behavior in different compositions. For example the template composition

```plaintext
primary_stmt :=
    component_decl | combine_decl | convert_decl
    | prepare_decl | template_decl | augment_decl
    | pair_decl
component_decl :=
    "component" '{' comp_stmt_list ';' '}' ID
combine_decl :=
    "combine" '{' comp_stmt_list ';' '}' ID
convert_decl :=
    "convert" '{' comp_stmt_list ';' '}' ID
prepare_decl :=
    "prepare" '{' comp_stmt_list ';' '}' ID
comp_stmt_list :=
    comp_stmt_list ';' comp_stmt | comp_stmt
comp_stmt :=
    "filename" '=' ID ';' | "function" '=' FUN
template_decl :=
    "template" '{' temp_stmt_list ';' '}' ID
temp_stmt :=
    "component" '=' ID ';' | match_stmt ';'
match_stmt :=
    "match" '(' FUN ',' FUN ',' FUN ')' | "match" '(' FUN ',' FUN ',' FUN ',' ID ')'
augment_decl :=
    "augment" '{' aug_stmt_list ';' '}' ID
aug_stmt :=
    "component" '=' ID ';' | replace_stmt ';'
replace_stmt :=
    "replace" '(' FUN ',' FUN ')' | "replace" '(' FUN ',' FUN ',' ID ')' 
pair_decl :=
    "pair" '{' pair_stmt_list ';' '}' ID
pair_stmt :=
    "component" '=' ID | "prep" '=' ID
```

Figure 4. The core PPCL grammar. Repetitive and obvious rules were cut out for brevity.

merges the components’ periodic functions. An augmentation however may be done in helper functions which are called from the periodic loop.

6.2.2 Compositions

For each composition in the methodology there is an associated keyword. Each keyword corresponds to a unique type in the language which defines the attributes particular to the composition. Names are given to each composition object which represent the newly generated component and can be used in later compositions within the same PPCL file.

- **Template** The template composition involves merging two components into a single component. We achieve this by merging the two periodic functions of each component. Often the two component loops share some similar function calls, for example a network send or target member selection. These duplicated calls can be merged into a single call in the final component, reducing overhead and simplifying the code. We use this idea to define the notion of “merge points”. Merge points are the function calls that are shared
Figure 5. A PPCL file for the SWIM template composition and an augmentation of SWIM using topology awareness (topo-SWIM). This example shows the components dping, epidemic, and topology-aware. Included in the template specification is the merge point wait_for_msg which uses the combine-wait combine type to complete the filler code. The template also has shared variables for the member list and message buffer.

between the two functions. The merge points are specified in PPCL with the match statement. Each function is delimited into blocks based on the merge points. These blocks are then combined around the merged function calls to generate the final result. An illustration is given in Figure 6.

A difficulty arises in the situation that the two merge points have different function signatures. Some arbitration must be done to determine which arguments from the original function calls are passed to the final function call. Filler code must be supplied by the user. The PPCL combine type provides a means by which a designer can specify this filler code in PPCL. An implementation of the combine function accepts the parameters from the original function calls and generates parameters for the new call.

• Augment The augment composition replaces some behavior in a component with new behavior from another component. Each augment type contains the name of one target component and any number of replace statements. A replace statement gives the name of the function call to replace and the name of the replacement component. The PPCL compiler decides where to execute the augmentations based on the original component specifications. In each function listed in the target component the function call in the replace statement will be replaced with a new call to the first function listed in the replacement component. Filler code to convert the parameters of the original call into parameters for the new call can be specified with the convert type.

• Pairing The pairing composition joins a proactive and a reactive protocol. The reactive protocol is grafted into the proactive via a specified “access point”. The compiler uses the function declared in the reactive component. The function is called once for every execution of the proactive periodic loop. The prepare type allows a designer to specify a function that gathers the arguments pass to the access point.

7 Additional New Compositions covered by the Methodology

Figure 2 showed probabilistic protocols that are generated by our described methodology. In this section we summarize additional protocols for wireless ad-hoc/sensor networks that are well-known and can be retroactively generated by the same methodology. Pseudocode can be found in the appendix.

Directed Diffusion Directed Diffusion is a routing protocol for sensor networks [17]. In diffusion nodes periodically advertise interest for certain data. Sensors that match the interest respond with the data. Nodes react to incoming data flow by “reinforcing” certain favorable routes. Di-
rected Diffusion demonstrates the *pairing* composition. The two components of directed diffusion are (i) a dissemination piece which spreads the interest and (ii) a reactive protocol which responds with reinforcement information. An optimization of the pairing composition is to piggyback reinforcement messages on interest packets.

**AODV with Gossiping** AODV [22] is a reactive protocol for routing in ad-hoc networks which itself is a composition of a dissemination and failure detection block. AODV uses a broadcast dissemination component to send route request (RREQ) messages. A failure detection protocol discovers when an existing route breaks. In [13] AODV is augmented with a gossip based dissemination protocol. The augmented AODV reduces end-to-end delay and improves packet delivery ratio without increasing route length.

**SHARP** SHARP is a protocol created with the pairing composition [23]. SHARP is a pairing of AODV and a proactive Zone Routing-like Protocol [14]. The proactive protocol is a composition of a dissemination protocol (builds the zone), a failure detection protocol (detects and notifies of broken links) and a recovery protocol (rebuids routes).

### 8 Implementation Evaluation

In this section, we experimentally study the PPCL and the methodology. We choose the problem of maintaining weakly consistent membership in a distributed group [5]. Specifically, we wish to study the following hypothesis.

A. Generated protocols function like hand-written protocols.
B. Properties of components truly are preserved.
C. The PPCL compiler does not add extra complexity to existing code.

We first use PPCL to generate a SWIM-like membership protocol from individual components, and then compare its performance with the original SWIM protocol of [5]. We created two stand-alone components for Uniform Epidemic and Distributed Ping. The implemented components are consistent with our assumptions: each protocol has a periodic program loop that executes the protocol.

Secondly, we make the above generated SWIM to be topologically aware. This is achieved by augmenting the above code (through PPCL) with a topology aware selection scheme. The scheme selects a node with the probability distribution function \( \frac{d}{p} \) where \( d \) is the distance to the remote node. The PPCL file used is in Figure 5.

Thirdly, we calculate several code complexity metrics on the generated code.

The generated protocols were evaluated in a simulated environment running in Windows XP on a Pentium 4 3GHz machine with 1GB RAM. Topologies from GT-ITM [4] were used to arrange the nodes. For the experiments in Figures 7, 8 the same topology was used for all runs. This topology had 4 transit domains with 4 stub domains each. The stub domains had average 8 nodes with .5 edge probability. The experiments of Figure 10 were done on two different topologies each of 135 nodes, 3 transit domains, 5 nodes per domain, 3 stubs per transit and 3 nodes per stub. All edge probabilities were .5.

**A.** We study the behavior of the generated SWIM, and explain why it is equivalent to that of the original SWIM protocol. We measured the times failures were first detected, update dissemination time, and resilience to dropped messages. The protocol was run for a fixed number of rounds on varying sizes of groups. At a certain round one member was killed. All other nodes remained active for the duration of the run. For each group size we ran two different trials on different topologies. The minimum detection time amongst all group members was taken and is shown in Figure 7. The detection times do not vary with system size, which matches with the original SWIM protocol.

The same experiments were used to measure rate of update dissemination. Figure 8 shows the results found by averaging the detection time of each node in the group. The points show individual detections and the plot shows the average. We observe a slow and sublinear growth, which is also consistent with the observations about the original SWIM system.

Finally, we measure the false positive rate (false failure detections of non-faulty group members). Members were allowed to join sequentially, while each message was dropped with some probability \( f \). The experiment was done with and without an optimization called the suspicion mechanism (which is used to reduce the rate of false positives by increasing the detection time) - this mechanism was present in the Distributed Ping component and hence inherited. Figure 9 shows the results for a value of \( f = .20 \). We observe that the generated protocol can be used to tolerate as much as 20% loss rate, when used in conjunction with the suspicion mechanism.

**B.** A new protocol, topo-SWIM was also created using PPCL. topo-SWIM is an augmentation of the basic SWIM protocol that uses a topology scheme to select target members. To study the effectiveness of topo-SWIM we counted the number of messages to pass through each router in the system. We ran the same experiment as for failure detection for 100 nodes but added 10% packet loss. We ran the experiment on two different topologies. The results for both topologies are included in Figure 10 with routers 0-14 from the first topology and 15-29 from the second. Some routers received no packets in either topo-SWIM or basic SWIM (0 height bars in the figure). We compared the message count at each router under topo-SWIM with the count under basic SWIM. Across all routers, topo-SWIM sent 95.3% the number of messages sent by basic SWIM. Loads are reduced by up to as much as 14% on some individual routers. The small
Theoretical Minimum
First Detection
2
3
3.5
4
0
0.5
1
1.5
2
2.5
3
3.5
4
group size
rounds

Figure 7. The time of first detection of failure. For each group size the first failure detection amongst all nodes is plotted. Values are given in rounds after the failure occurred.

Figure 8. Failure update dissemination time. Detection time at individual nodes is compared to the time of first detection.

savings are due to the short diameter of the underlying network; with a larger number of stubs and transits, the router load savings would have been higher.

C. After running the simulations we also analyzed the complexity of the source code. This was important to show that the code generated by PPCL is understandable and usable by a human reader. We chose two of the more commonly used static metrics for code complexity: cyclomatic complexity [21] which measures a code’s stability and Halstead volume [15] which measures lexical complexity. The metric values for each of the described components are in Table 1. Ideally the complexity of composed protocols would be no more than the sum of the components’ complexities.

Under template composition the complexity increased slightly. This slight increase is expected due to the code changes affected by the template composition, and is acceptably small. Augmentation adds little to no complexity which is also expected since it does not involve major code changes.

Overall these results show that the code complexity of the generated protocols is not considerably worse than the original component implementations.

9 Conclusion

We have described a simple and clean methodology consisting of building blocks (protocols or strategies) and composition rules. The methodology is powerful enough to generate a class of protocols for large-scale Internet-based process groups [5, 8, 9, 10, 11, 12], as well as for wireless ad-hoc/sensor networks (Section 7). We have then presented a high level language called PPCL that instantiates this methodology. PPCL can be used to specify existing
source code as components, and allows a designer to automatically generate code for composed protocols. We have argued how the methodology and PPCL are retroactive (i.e., generate already existing protocols), and progressive (i.e., generate new protocols). Our experiments show that generated protocols have similar behavior to corresponding hand-designed protocol, and that compositions preserve scalability and reliability of the components.

**Future Work:** PPCL could be extended to support property inferring. For each component, its properties would be specified in PPCL. Using the given properties the compiler would reason about the properties of the generated protocol.

**References**


A Pseudocode Examples

T: protocol period.

DIRECTED-DIFFUSION
while (interested)
  msg ← new data_interest
  DISSEMINATE(msg)
  WAIT(T)
repeat

Figure 11. Pseudocode for Directed Diffusion interest propagation.

DD-RECV-DATA(data)
if IS-HIGH-QUALITY(data.src)
  pend − rf ← new reinforcement(data.src)
  SEND pend − rf
endif

Figure 12. Pseudocode for Directed Diffusion reinforcement component.

DIRECTED-DIFFUSION-PAIRED while (true)
  msg ← new data_interest
  if pend − rf != null
    then
      msg ← PIGGYBACK(msg, pend − rf)
      pend − rf ← null
    endif
    DISSEMINATE(msg)
    WAIT(T)
  repeat

Figure 13. Pseudocode for main loop of Paired interest propagation and reinforcement components from Directed Diffusion.

AODV-REQUEST-ROUTE(dst)
  route ← rtable[dst]
  if route == null
    then
      msg ← RREQ(dst)
      FLOOD (msg)
    endif

AODV-NODE-FAILURE(node)
  foreach route r that contains node
  do
    msg ← RERR(r)
    FLOOD (msg)
  done

Figure 14. Pseudocode for AODV route request and failure notification.

/* Ms: the current member. */
/* N: estimate of group size */
/* b: gossip fanout */
/* ViewMs: Membership List at member Ms */

UNIFORM-EPIDEMIC(multi − cast − msg)
for logN gossip rounds
  for i ← 1 to b
    Mtarget ← SELECT-RANDOM-NODE(ViewMs)
    UNICAST(Mtarget,multi − cast − msg)

Figure 15. Pseudocode for uniform epidemic.
AODV-REQUEST-ROUTE\((dst)\)
route ← rtable[dst]
if route == null
then
    msg ← RREQ\((dst)\)
    UNIFORM-EPIEDMIC(msg)
endif

AODV-NODE-FAILURE\((node)\)
foreach route \(r\) that contains node do
    msg ← RERR\((r)\)
    UNIFORM-EPIEDMIC(msg)
done

Figure 16. AODV augmented with uniform epidemic.

ZONE-CONSTRUCTION
while (true)
    msg ← DAG – construction\((self)\)
    FLOOD(msg)
    WAIT\((reconstruct – interval)\)
ZRP-RECV-DATA\((msg, dst)\)
next hop ← routes[dst]
if next hop! = null
then
    UNICAST\((next hop, msg, dst)\)
else
    FLOOD\((msg)\)
endif
ZRP-RECV-CONST\((target, src)\)
routes[target] = src

Figure 17. A Zone Routing Protocl-like algorithm.

SHARP-ZONE-CONSTRUCTION
while (true)
    msg ← DAG – construction\((self)\)
    FLOOD(msg)
    WAIT\((reconstruct – interval)\)
SHARP-RECV-DATA\((msg, dst)\)
if dst == myaddr
then
    DELIVER\((msg)\)
else
    next hop ← routes[dst]
    if next hop! = null
then
    UNICAST\((next hop, msg, dst)\)
else
    AODV-FWD\((msg, dst)\)
endif
endif
SHARP-RECV-CONST\((msg, src)\)
routes[msg.target] = src
if -- msg.ttl > 0
then
    FLOOD\((msg)\)
endif

Figure 18. SHARP protocol which is a pairing of a Zone Routing Protocol-like algorithm and AODV.