Chameleon - A Group Communication Framework for Smartphones

Alex Dvinsky  
Electrical Engineering Department  
Technion  
Haifa 32000, Israel

Roy Friedman  
Computer Science Department  
Technion  
Haifa 32000, Israel

April 22, 2013

Abstract

This paper reports about our experience in designing and developing Chameleon, a highly portable and adaptable group communication framework for smartphones. Chameleon owes its level of portability to several design choices, including: (a) a layered architecture, where the headers of each layer have a standard XML-based format, enabling automatic, error resistant generation of efficient serialization code in any platform, (b) reliance only on the J2ME library, which serves as least common denominator for Java dialects and facilitates automatic translation to .NET, (c) having flexible membership models, and (d) supporting multiple concurrent protocol stacks.

Through a single code-base, Chameleon is currently available as an open source project for J2ME, J2SE, Android, .NET CF, and .NET. Chameleon is easily extendable and is bundled with tools, configurations and third party code tuned in a way that lifts some of the burden normally associated with multi-platform development for smartphones. Both the header generation from XML and automatic translation to .NET features of Chameleon are readily available to any application that is based on it.

Chameleon’s threading model separates between execution of internal layers and application’s code and by that protects one from the other. As we describe in the paper, it simplifies layers’ development and allows the protocol stack to easily block application calls when this is required by internal algorithms. Additionally, this model simplifies testing and an extensive testing framework is supplied along with Chameleon, which is also usable for testing of application-specific layers.

1 Introduction

During the last few years, smartphones have become ubiquitous devices, frequently used not only as phones, but also as game consoles, means for capturing and consuming media content, navigational assistants and more. Yet, often such services are provided either as a stand-alone application or in a client-server mode, when the server is accessed via the Internet. Such mode of operation can be impractical to users when an Internet connection is not available or is expensive. An application utilizing the connectivity of an available local WiFi network, or forming an Ad-Hoc connection with peers, would often be more convenient to users. In particular, there are various situations and applications in which in any case all that is needed is connectivity with other nearby devices. These

*Partially supported by the Israeli Science Foundation grant number 1247/09
include streaming media between devices, sharing family pictures and videos, students chatting with each other during class, etc. In these, a symmetric decentralized form of communication is much more economical and natural than a client/server model.

However, when not relying on a constantly available server, achieving otherwise trivial properties, such as monitoring connectivity status, message ordering and reliability becomes a complicated task. Traditionally handling of this complexity is delegated to a group communication middleware. Group communication toolkits usually provide a notion of a view representing live and connected members with varying levels of synchronicity properties. Group communication supplies reliability and event ordering, and creates an illusion of synchrony using distributed software algorithms, rather than relying on a central server.

Smartphone is a loosely defined term, which covers a great variety of devices with different hardware specifications and operating systems running on them. An application developer is usually interested in covering as many of the available platforms as is feasible, to attract the maximal number of potential users. However, development tools, languages and flows for these platforms are often incompatible with each other, making such development difficult.

In addition to the heterogeneous nature of smartphone operating systems, a considerable number of mobile devices, not powerful enough to be called smartphones exist on the market. According to some studies, their number is as much as 50% even in the USA. It is reasonable to expect that this percentage is much higher in less developed countries. For such devices, often, J2ME is the main, or even the only platform they support.

In this paper we present Chameleon - an open source, multi-platform group communication framework targeted at mobile devices, but not limited to them. It offers support for development of infrastructure-less, connected applications and lifts some of the burden associated with multi-platform development. Chameleon is intended to operate in a local or an ad-hoc type of network. Chameleon is available for both mobile platforms (J2ME, Android, .Net Compact Framework) and PC platforms (J2SE, .Net). Much of its portability and interpretability comes from having an XML specification for its message headers, enabling standard cross platform serialization. Also, Chameleon’s code is based on J2ME, which seems to be a least common denominator for most Java platforms, and enables its automatic porting to .NET platform.

As detailed in this paper, Chameleon follows the modular approach of building a user-space protocol stack from micro layers, thus allowing great flexibility for application developers as well as many extendibility options. Further, the protocol stack of Chameleon can be multiplexed to allow several membership models and QoS configurations in parallel and in particular, it is not restricted to the virtual synchrony model. Chameleon also offers an extensive testing framework, which enables easy test writing and testing in a close-to-production fashion for new layers. Chameleon’s threading model eliminates most group communication threading related concerns from the application developer and from the layers’ developer. Finally, Chameleon encourages the development of multi-platform distributed applications by offering the needed set-up and tools for generating application level header objects that are easily interpretable into its messages along with the needed set-up to perform easy translation of Java applications based on Chameleon to .Net CF compatible code.

The rest of the paper is structured as following: Related work is surveyed in Section 2. Section 3 gives a high level overview of Chameleon’s components, details its header generation capabilities, and...
discusses platforms supported by Chameleon and explains the porting procedures. Section 4 dives into finer details and explains the functionality of the main components, their assumptions and guaranties; it lists the available layers and provides a short description of each. Section 5 outlines basic usage patterns from the perspective of an application developer. Section 6 presents the benchmarks that were performed by us on Chameleon on real devices including the set-up of the system, flow of the tests and the results. Section 7 presents a few sample applications that were written based on Chameleon. The main paper is concluded in Section 8, with some additional technical details appearing in the appendices. Appendix A contains examples of XML header files and application code. Appendix B gives an overview of the testing framework.

2 Related work

The area of group communication has been extensively studied [16]. The first implementation of group communication was Isis [15]. Many others have followed and the area is still evolving. Initial group communication systems were monolithic [11, 13, 15, 32] in the sense that the entire system was designed as an integrated software. Later on, modular systems that are composed of micro-protocol layers have been developed, such as [3, 9, 25, 37, 38]. In this work, we have also chosen the layered approach due to its obvious software engineering benefits and in order to provide the application with a fine grain control over the protocol (quality of service) it uses. Yet, unlike previous group communication systems we are aware of, Chameleon specifically targets smartphone devices' (and possibly other mobile embedded platforms), starting with the limited J2ME platform. Also, our system goes a step forward in terms of its portability and inter-operability, due to our use of standard XML-based headers and serializers at each layer, which enable us to support so many platforms with a single code base for 99% of the functionality.

Most traditional group communication systems have been developed for LAN environments, in which network problems and message losses are rare, and the behavior of the network is usually symmetric and transitive. Further, the main applications these systems were developed for involve state replication, for which strong consistency semantics is highly beneficial and often even critical. Consequently, these systems usually support some variant of the virtual synchrony model [21] (or Paxos style replication [27]). Other systems targeting wide area networks, such as Spread [12], have somewhat relaxed this model due to the high cost associated with providing such strong semantics over the Internet.

In Chameleon, we support both a weak asymmetric view semantics and virtual synchrony. The former is due to the fact that in applications formed by ad-hoc collections of mobile phones, disconnections and network interruptions are likely to occur frequently. Hence, providing a strongly consistent view semantics may be impossible and many of the envisioned applications, such as file sharing/streaming, chatting, and game playing can deal with weaker view semantics. However, for compatibility with traditional models, and in order to facilitate multiparty ad-hoc transactions, Chameleon also includes a virtually synchronous membership layer. In particular, Chameleon supports having multiple concurrent protocol stacks of varied semantics (including a different membership view in each such stack).

Following the Bimodal paper [14], multiple works on scalable reliable dissemination and scalable membership services have been published, e.g., [19, 20, 24, 26, 28, 29, 31, 39]. These works attempt to limit the communication fan-in and fan-out degree of each node either through random (but not necessarily uniform) gossip, or by flooding messages along the arcs of a logical overlay (deterministic
or random). At the moment, none of these protocols is implemented in Chameleon, since the typical application we envision only includes a few (and up to a few dozen) devices that are likely to be either on the same LAN segment, or connected through an ad-hoc network. However, given that our basic membership model is asymmetric and due to the layered design of the system, adding any of the above protocols should not be difficult.

Isis$^2$ is another recent group communication system [5]. It mainly targets large data centers, and thus includes support for wide area discovery, efficient scalable reliable dissemination, and both virtual synchrony/Paxos semantics and a weaker asynchronous membership model. Isis$^2$ also includes good integration with Web services and transparent object replication.

Baldoni and Prakash have presented an architecture for group communication in mobile networks [31]. Yet, their work is a conceptual one, and lacks a real implementation. Other conceptual works that surveyed the implications of ad-hoc networks on group communication include, e.g., [17, 23].

Other systems for ad-hoc networks have been implemented over simulators, such as [30]. JazzEnsemble is a port of Ensemble [25] to ad-hoc networks including support for Byzantine failures [18]. Yet, it is written in the OCaml language, restricting its usage to platforms that support OCaml.

3 High Level Overview

3.1 Main components

As indicated above, Chameleon is a layered group communication middleware such that the message headers corresponding to each layer have an XML signature. As such, it mostly consists of the following components (see Figure 1):

Layers: The building blocks of a protocol semantics, each of which is responsible for some relatively simple logic, such as heartbeat, broadcasting or duplicate filtering layers.

Protocol Stack: Combines individual layers. The properties of the protocol semantics are defined by the layers used within a stack and their relative order.

Events: Hierarchy of objects through which layers communicate with each other and the rest of the world.

Headers: Data containing objects generated from XML. These objects can be added to or read from events representing outgoing messages or incoming messages.

EventChain: The entity controlling the flow and handling execution of the events by layers of the stack.

Serialization Utils: Set of serialization routines used by the framework and available to the applications for header serialization.

Serializers Generator: A utility that generates serializer classes for generated header classes.

J2ME Data Structures: Several GNU Classpath Java data structures adapted for use in J2ME.
3.2 Headers

Many layers in Chameleon define custom headers, which are added to outgoing messages and used to communicate with respective layers in other stack instances. These headers in Chameleon are defined by XML files in a standard W3C XML Shema format [10]. These XML files are compiled to Java classes with JiBX [6] and have custom serializers generated for them. Such fast, error resistant flow not only simplifies the development and extension of Chameleon, but can also help an application developer that is likely to need application-level headers as well (see Section 5.7 for an example). Besides, these serializers are platform independent and a header serialized by one node can always be read by another, as illustrated in Figure 2.

3.3 Platforms

As mentioned above, Chameleon is a multi-platform framework. It is compatible to J2ME, J2SE, Android, .Net and .Net Compact Framework. The Minimal requirements for these platforms are extremely modest:
<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimal version (and up)</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2ME</td>
<td>MIDP 2.0, CLDC 1.1</td>
</tr>
<tr>
<td>Android</td>
<td>Version 1.6</td>
</tr>
<tr>
<td>J2SE</td>
<td>Version 1.3</td>
</tr>
<tr>
<td>.Net CF</td>
<td>Version 2.0</td>
</tr>
<tr>
<td>.Net</td>
<td>Version 2.0</td>
</tr>
</tbody>
</table>

The absolute majority of Chameleon’s code is written in the Java language compatible to J2ME. J2ME, being a very restricted platform, suites well for the role of lowest common denominator, since most code written for it is immediately usable with J2SE and Android. This minimalistic nature required implementing otherwise commonly available functionality, such as any data structure more complex than Array. While this has required some effort, it turns to our advantage when .Net is considered. Such little dependency on advanced Java platform allows Chameleon to be translated automatically to C# by [8] with only little configuration tweaking and one-time implementation of a relatively small set of functionality that is missing in .Net Compact Framework (see [Figure 3]). Chameleon hides from the application developer the diversity of network operation syntax on various platforms and, to some extent, hides their limitations.²

In addition to being available for potential target platforms, Chameleon also simplifies multi-platform development by making its porting approach and tools easily available for an application developer. If the same principles are followed during the development of an application, then porting its network related modules to all of the above platforms becomes a breeze. The code is either immediately compatible, as will be the case with all Java-related platforms, or can be mostly automatically translated, considering that much of the needed functionality that is missing in .Net is already implemented and comes built-in with Chameleon (more about it in 3.3.2).

![Figure 3: Chameleon and platforms](image)

### 3.3.1 Port to J2SE and Android

Though most of J2ME code is usable on the richer Java platforms, it cannot provide full platform independence as the syntax of such operations as opening and closing sockets is different on each and every platform. For this, Chameleon includes platform-dependent parts that take care of such non-portable operations, which are hidden in the lowest parts of Chameleon behind generic interfaces.

²For example, in some J2ME implementations, one cannot have more than one socket opened at a time. Using Chameleon, overcoming this limitation and maintaining a seemingly constant connection with the whole group becomes a matter of setting a flag to the network communication layer.
3.3.2 Port to .Net

Porting of the Java code to .Net is done automatically with Sharpen. This tool runs using Eclipse’s Abstract Syntax Tree API to parse java source code and outputs it in C# syntax with configuration controlled mappings to overcome language incompatibilities. Generated header classes along with their serializers are translated as well. Then, auto-translated code is supplemented with platform-dependent parts and compiled by a .Net compiler. The same approach is easily extendable to Chameleon based applications and in fact was used by us to port several applications. Porting to .Net core (non UI) components of an application, which was developed using the same guidelines that were used to develop Chameleon and its demo applications, can be done in under 5 minutes. Only a reconfiguration of some properties is required while all the rest, even the type mapping, already comes with Chameleon.

4 Framework Internals

4.1 ID Generator

During system’s operation, an entity often needs to be given a statistically unique ID. For that purpose, Chameleon contains an ID Generator that is seeded during the protocol initialization with the 8 least significant bits of the node’s IP address and current timestamp. The ID Generator produces IDs that are 64 and 32 bits long. The high 8 bits of the produced ID always contain the 8 least significant bits of the node’s IP address and the rest are random.

4.2 Events

Events in Chameleon are structured in a hierarchy of classes with a common root. Layers identify events by checking their runtime types and sometimes process a whole branch of events, rather than treating the types individually. For example a layer can specify its behavior for all types of outgoing messages, not caring about their addressing modes. Each event can carry additional information. Each event has a randomly generated 64-bit long ID that allows to tell it apart from other events, potentially carrying the same information. The ID is produced by the ID Generator component. See Figure 4 for a complete hierarchy of events available in Chameleon.

4.3 AsyncExecutor

Chameleon contains an asynchronous invocation mechanism called AsyncExecutor. Each protocol stack, by default, has an instance of this mechanism, shared by all layers. AsyncExecutor contains a thread and maintains a list of registrations for execution. Registrations consist of a requested invocation time and a Runnable instance. AsyncExecutor will invoke the given Runnable at the requested time in the context of its thread. This mechanism is used for all periodic (timeout based) operations, such as heartbeats, resubmissions and delayed acknowledgements.

4.4 Event chain

An event chain contains a list of event handling requests and a processing thread. Each event handling request contains the event itself, the layer this event should be handled by and the direction of the event. Chameleon is designed in such a way that event handling is only done in the context
Figure 4: Events hierarchy
of event chain’s processing thread. Each protocol stack contains at least two such event chains. One chain for the stack’s internal functionality and the other for the application layer. Such a separation protects the protocol’s operation from the application and makes sure that events in the stack are always able to propagate.

The fact that most layers run in the context of a single thread removes the need for synchronization in them and simplifies their writing. Some layers, however, use additional threads for their internal operations. For example, when the network layer is waiting for messages in a blocking call, or an asynchronous invocation by AsyncExecutor. These layers might still require synchronization. See Figure 5 for an illustration of the threading model.

When events are added to the chain, the result depends on the thread the operation is performed by. If the add operation is performed by the internal thread of the event chain, it means that it is done by a layer, responding to the last event that was taken from the chain. In this case, the added events will be put first in the list of events to be handled while preserving their internal order (see illustration in Figure 6). If the operation is performed by another thread, the events added are put last in the list of the events (see Figure 7). Such logic is required to preserve causality in events ordering.
4.5 Layers invocation

Most of the time layers either respond to events or produce them when triggered by other, external means. Events can be received from a neighboring layer, from above or from below. Once an event has been received, it belongs to the layer and it is up to this layer to decide what to do with it. The event can be discarded, passed on, consumed etc. Each layer generally acts on some events and has no interest in other events, which are simply passed on in the same direction.

There are two possible ways for a layer to pass an event. Either directly, by invoking the appropriate handler of a neighboring layer, or to place the event into the event chain and let the chain handle the order. The former is simpler and somewhat faster, but should be used with great deal of caution as it might disrupt the intended ordering of events or violate other layers’ threading assumptions (if called from a non-event chain thread). The latter option is somewhat slower, but is completely safe.

4.6 Layers

Layers are the protocol building blocks. Multiple layers are available as part of Chameleon and while for most needs the provided ones should be enough, an application developer can easily implement an additional layer and use it in a stack along with core layers if such a need arises. Below, we provide an overview of the layers available in Chameleon. For a complete understanding of the available functionality, in addition to their high level description, one needs to dive into details such as

- What is the configuration of the layer
- Which events the layer responds to and from which directions
- Which events it produces and in which directions they are sent
- Which events it consumes
- What headers does it add to the messages
- The threading assumption of the layer. Does it only run in the context of the event chain thread or there are others?

These aspects, however, are too fine for this paper and are thoroughly detailed in the system’s documentation.

<table>
<thead>
<tr>
<th>Layer name</th>
<th>Description</th>
</tr>
</thead>
</table>

10
<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDPNetworkLayer</td>
<td>Communicates with the outside world using UDP.</td>
</tr>
<tr>
<td>TCPNetworkLayer</td>
<td>Communicates with the outside world using TCP.</td>
</tr>
<tr>
<td>FramingLayer</td>
<td>This layer restores message boundaries if underlying infrastructure is stream based, as the case in TCP. If TCPNetworkLayer is used, this layer is mandatory and is to be placed just above it.</td>
</tr>
<tr>
<td>MessageTranslationLayer</td>
<td>Translates between events of type outgoing/incoming message and events of type outgoing/incoming data. The outgoing/incoming data event is basically a buffer with an IP address, while message typed events contain slightly more data, such as the message ID and the ID of the sender. This is a mandatory layer and is to be placed just above UDPNetworkLayer or FramingLayer.</td>
</tr>
<tr>
<td>StackIdentificationLayer</td>
<td>Tags all outgoing messages with a preconfigured ID and filters all incoming messages whose ID does not match the configuration.</td>
</tr>
<tr>
<td>OwnMessagesFilter</td>
<td>Filters out incoming messages sent by the protocol instance it resides in. Such messages can be produced, for example, by UDP broadcast.</td>
</tr>
<tr>
<td>CrcCheckLayer</td>
<td>Adds to the outgoing messages a header containing their CRC and filters incoming messages whose CRC does not match one in the header. This layer can be used if underlying UDP does not use the (optional in UDP) checksums.</td>
</tr>
<tr>
<td>MulticastByUnicast</td>
<td>The layer substitutes all outgoing multicast and broadcast messages with a separate message sent to each of the intended recipients. For a multicast message, the recipients are listed in the message itself. In case of a broadcast message, the list is taken from the currently known inconsistent view.</td>
</tr>
<tr>
<td>MulticastByBroadcast</td>
<td>The layer substitutes all outgoing multicast and broadcast messages with a single message sent to preconfigured UDP broadcast address. In case of a multicast message, also a list of intended recipients is added. When receiving, if current protocol instance is not listed in the recipients list, the message is discarded.</td>
</tr>
<tr>
<td>IdleDetectingLayer</td>
<td>The layer expects to hear from each member of the known inconsistent view at least once in a configured period of time, otherwise the member is declared idle and upper layers are informed. Often, it is used in conjunction with HeartbeatLayer or UDPBroadcastDiscoveryLayer.</td>
</tr>
<tr>
<td>HeartbeatLayer</td>
<td>Sends a broadcast message out once every configured interval of time. Consumes incoming heartbeat messages.</td>
</tr>
<tr>
<td>UDPBroadcastDiscoveryLayer</td>
<td>Sends out a message targeted at the configured UDP broadcast address once every configured interval of time. Consumes incoming broadcast messages. If this layer receives a broadcast message from a stack instance which is not a member of the current inconsistent view, it updates upper layers about a newly discovered member.</td>
</tr>
<tr>
<td>Layer</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AckNackLayer</td>
<td>Remembers every outgoing message and expects to receive an acknowledgement on each within a configured period of time. If such an acknowledgement was not received, the layer informs upper layers that the message was not delivered to its recipient(s) and attaches the list of recipients that did not acknowledge the message. If a target recipient of a message leaves the current inconsistent view, it is no longer expected to ACK messages and no non-delivery event will be generated for them. Upon reception of a message, this layer acknowledges it. It also has the ability to delay the acknowledgement until either a configured timeout passes, or a configured number of unacknowledged messages accumulates, the earlier of the two. If the originator of a message leaves current inconsistent view, its messages are no longer being ACKed.</td>
</tr>
<tr>
<td>ReliableDeliveryLayer</td>
<td>Resends messages that were informed by AckNackLayer as non delivered.</td>
</tr>
<tr>
<td>NoDuplicatesLayer</td>
<td>Filters out duplicate messages. Duplicates are detected by a message ID. IDs of the received messages are kept for a configured period of time.</td>
</tr>
<tr>
<td>SequencingLayer</td>
<td>Maintains a message counter and attaches an increasing counter to outgoing messages. This layer also enforces sender FIFO order on the received messages. It must be used with the reliable communication infrastructure layers.</td>
</tr>
<tr>
<td>MuxLayer</td>
<td>Splits the stack above itself to separate paths, each topped with its own ApplicationLayer. See Figure 9.</td>
</tr>
<tr>
<td>InconsistentViewManagementLayer</td>
<td>Maintains the inconsistent view. Adds discovered nodes to the view and removes idle nodes from it. Responds to requests from an application to merge views. Sends appropriate inconsistent view change events in both directions.</td>
</tr>
<tr>
<td>VirtualSynchronyLayer</td>
<td>Provides a notion of strong virtual synchrony as defined in [21]. Handles coordinator elections, messages retransmission and consistent view management for upper layers.</td>
</tr>
<tr>
<td>TotalOrderLayer</td>
<td>Provides total ordering of messages in a virtually synchronous environment. The messages are ordered by the current consistent view coordinator, which periodically sends out order information to the rest of the view.</td>
</tr>
<tr>
<td>LeakyBucketFlowControlLayer</td>
<td>Allows messages to pass through at a configured rate. If messages arrive at a higher rate, they are delayed. In such cases, an event notifying about it is sent up the stack. Once the message will be sent, an event about this too will be generated upwards.</td>
</tr>
<tr>
<td>ThreadSwitchLayer</td>
<td>This layer disconnects upper layers from the event chain used by the lower layers. This allows the application layer to run in its own event chain and protects the rest of the stack from it.</td>
</tr>
<tr>
<td>BlockingLayer</td>
<td>Contains the functionality of ThreadSwitchLayer, but additionally can block the thread that requested handling of an outgoing message event. This is used to ease the flow control implementation for the application and to support the virtual synchrony mechanism, which requires blocking of outgoing messages at certain points in time.</td>
</tr>
</tbody>
</table>
4.6.1 Sample Protocol stacks

Figure 8 illustrates several possible combinations of protocol stacks. As can be seen, the minimal protocol stack includes, in addition to the application, the UDPNetwork layer for accessing the network, the MessageTranslation layer that translates events into messages and back, and the ThreadSwitch layer that separates the control of the application from that of the protocol stack.

The “basic” stack in Figure 8 supports multicast and inconsistent view management, while the other stack provides in addition reliable message delivery with flow control. These stacks were used for our benchmarking reported in Section 6 below and are likely to be used by many envisioned applications. The top most layer in both is the DataCounting layer, which is used to log the amount of data traveling through the stack; it is optional, and only needed for debugging and benchmarking purposes, as was in our case.
5 Usage

5.1 Building a stack

There are several layers that must be present in a stack in order to conform to the operation model assumed by Chameleon. Such layers are the network layer (either UDP or TCP based), Message-TranslationLayer, ThreadSwitchLayer (or Blocking layer, the functionality of which supersedes that of ThreadSwitchLayer) and an application layer.

Most layers have some kind of dependency on their stack-wise surroundings, so not every (not even the majority of) possible layer combinations produce meaningful structures. Yet, developers need not be aware of such fine internal details, but rather can start with a basic stack from an example and adapt it to one’s needs.

The protocol stack is constructed bottom-up, always starting with a network layer and always finishing with an application layer. Chameleon also allows building fork-like multi-protocols, when at one specific place a split point is set and several separate paths, each topped by its own application layer, are built atop of it as shown in Figure 9. Such constructs can be used to easily build protocols with multiple QoS configurations, including multiple membership models for various aspects of the application.

5.2 The application

The application needs to implement the top-most layer of the stack – the ApplicationLayer. This layer is responsible to produce outgoing events and respond to incoming ones. In most cases, these events are carrying application messages, as most of the other events are consumed by the underlying layers. Examples of non-data-carrying events that can be handled by the application are view changes and delivery acknowledgements. Chameleon contains an abstract class implementing the default behavior, so an implementation of this layer can be just a matter of a few lines of code. An example in subsection A.2 shows the basic usage.

---

In fact, this is not a hard requirement. One can place layers above the application layer as well. Yet, in this case one has to be aware of the threading implications of such placement.
5.3 Initialization

The stack needs to be initialized prior to its usage by calling the `init()` method of the Protocol-Stack instance. This will spur service threads, activate an internal event chain and pass a direct initialization event to all layers.

5.4 View types (membership models)

The current implementation of Chameleon supports two types of views: consistent and inconsistent. The application can choose the one that better suits its needs and build the stack accordingly. The inconsistent view is basically a combined output of member discovery with a failure detector. That is, it represents the current local estimation of the group of live members. This view is maintained by the InconsistentViewManagement layer.

Consistent view is a virtually synchronous view [16, 21], maintained by the ConsistentView-Management layer. Virtual synchrony is the membership model supported by most LAN oriented group communication toolkits. Together with a total ordering layer and safe delivery layers, it enables obtaining true state machine replication [22, 36]. Thus, it is useful for applications that require strong safety properties. Further, even for applications that can be developed with weaker membership models, it greatly simplifies the application design and code. On the other hand, it comes with a price: That is, each view change becomes a communication intensive operation, and also introduces a short performance hiccup [21]. Further, each view change requires global coordination, and must occur on each node’s departure or arrival. Especially in a network of mobile phones, these are likely to take place frequently. Consequently, it is likely that developers of most mobile phone applications would prefer to avoid it.

One may ask why bother with a virtually synchronous membership layer. The reason is that it can be useful, e.g., when a plurality of phones wish to perform a transaction. In this case, Chameleon enables them to create a virtually synchronous view with only themselves in it, and then benefit from its strong safety properties that are needed in such a case.

5.5 Messaging

To send a message, the application needs to construct an instance of the Data class and pass it to the application layer. The abstract level of the application layer contains various methods for message passing, which mostly differ by its addressing mode - broadcasts, multicasts or directed at specific address/view member. For each sent message, a random 64-bits long ID is generated by Chameleon’s ID Generator. Later on, events related to that message will contain its ID. To receive a message, the application overrides the appropriate methods of the abstract application layer.

5.6 Event handling

Handling of additional events, such as view change, is done in a similar way. That is, an appropriate method is overridden by the application layer.

5.7 Application level headers

As mentioned before, generation of headers from XML along with their serializers is helpful both for Chameleon development and maintenance as well as for application development. Consider a simple
chat application with multiple virtual chat rooms. The application has two types of messages:

- Chats sent by users.
- Control messages notifying about users joining or leaving a room.

A developer of such an application would only need to write XML files similar to the ones in subsection A.1 and compile them using the provided tools. In his application, the code responsible for sending a chat message and joining a chat room could be written as in the example in subsection A.2.

6 Benchmarks

Chameleon’s performance was benchmarked with real devices and the results are presented in this section. Performance numbers by themselves are of limited use since the results depend on many factors external to Chameleon such as hardware performance, quality of the underlying infrastructure implementation and wireless noise conditions. Therefore, when possible, a benchmark of a bare UDP scenario was added to serve as a baseline.

6.1 System set-up and test flow

The benchmarking environment consists of a master process running on a desktop machine and several phones, all connected to the same wireless ad-hoc network. The master is running a J2SE version of the master benchmarking application, whereas phones run the slave versions most suitable for their OSes. The master machine is connected to a relational database that contains definitions of test batches and stores the results.

The life cycle of the slave application is illustrated in Figure 10. Slave benchmark devices initialize an instance of the protocol stack with automatic discovery and reliable delivery features. This stack is used to pass control messages, namely receive test definitions from the master and submit the measured metrics to it.

Once the test definitions are received, each slave node shuts down the control stack and runs the test according to the received instructions. During test execution, various metrics are collected and once the test is over they are sent back to the master.

Each test specifies what protocol is to be used, and which load is to be transferred by it. The test is split to three major phases - initialization, benchmark, and finalization. The phases’ lengths are 5, 60 and 10 seconds respectively. During the initialization, protocol stacks and structures are initialized. During the benchmark phase, configured load is generated and sent through the stack. The finalization step is used to exchange application-level ACKs, needed for latency measurements. The test master receives reports from all the nodes, aggregates them and stores them in the DB.

6.2 Baseline

Figure 11 shows a performance comparison between a connection on a bare UDP socket serving as baseline and a very basic stack, with no reliability and only partial support for view management. Both tests were performed with two devices, a Samsung Galaxy SII and an HTC Desire, when one (Galaxy SII) is the sender and the other one is the receiver. Each point on the graph is the average of the respective value measured on several hundreds of messages. For the latency test, messages
of varying sizes are sent every 2 seconds. In the throughput test, messages of constant size (30,000 bytes) are sent over ever shrinking periods of time.

As can be seen, while bare UDP performs somewhat better, the difference is negligible in terms of the measured throughput and the difference in latency quickly fades as the message size grows.

6.3 Reliability

As shown in Figure 11, the sender’s throughput is higher than the receiver’s, meaning that messages are being lost. The problem becomes even graver when the load is highly non-uniform. Further, in Figure 12 on the left, we see how the throughput drops when the load contains bursts of multiple relatively short (1,000 bytes) messages and in Figure 13 the same phenomenon with bursts of long (30,000 bytes) messages. One obvious solution is to use TCP instead of UDP as the transport. Unfortunately, this is not always possible. Sometimes we need broadcast capabilities of UDP and sometimes TCP is just not supported.

To address this problem, Chameleon contains a set of layers intended to enable reliable communication over UDP. The most common of those being AckNackLayer and ReliableDeliveryLayer. The AckNackLayer will respond with ACKs on received messages and will generate up-going events notifying about messages’ acknowledgment status. The AckNackLayer is usable by itself, if an application is only interested in knowing whether a message was received, but to achieve reliable delivery it is easier to use it in conjunction with ReliableDeliveryLayer. ReliableDeliveryLayer resends a message upon reception of a non-delivery event so long as its recipient is still a member of the (inconsistent) view.

Adding these two layers to the stack solves the reliable delivery problem. However, under non-uniform load as before, it will increase memory consumption, since messages are kept longer in the system and require many resubmissions. Reshaping the load to be more uniform would allow many more messages to be submitted successfully and fewer resubmits would be required. Such shaping is achieved by adding a flow control layer to the stack. LeakyBucketFlowControlLayer is one such implementation, but many more could be added easily. LeakyBucketFlowControlLayer

---

As was the case with one of the J2ME platforms we tested.
Figure 11: Comparing performance of a basic stack with UDP socket baseline
only allows a message to propagate down the stack if the currently observed message throughput does not exceed a preconfigured threshold. For each delayed message, one up-going event is generated when the message is delayed and another one when it is released. As with AckNackLayer, the LeakyBucketFlowControl layer is usable by itself, if the application is coded to respond to delay/release events and adjust its sending rate according to it. If not, the stack could substitute its ThreadSwitchLayer with BlockingLayer. BlockingLayer effectively turns all send method calls performed by an application to blocking ones and these calls will be blocked until all delayed messages are released. Note though that such blocking still allows the application to receive messages.

Figures 12 and 13 demonstrate how using resubmission only causes the percentage of resubmitted messages to grow considerably and how the problem is solved by adding flow control. Note that in Figure 13, even with the longest burst size, the overall throughput is much lower than was measured in Figure 11, while the messages sent by the application are of the same size. This serves as evidence that the reason for the message loss and subsequent resubmissions is the non-uniformity of the load.

6.4 Throughput of broadcasts

In addition to throughput of direct communication between two devices, we also measure the throughput of broadcast. Chameleon supports two ways of message broadcast – either by unicasting the messages to all view members, or by using UDP broadcast. Here we present benchmarks of both. In case of broadcast by unicast with a single broadcasting source, its total available bandwidth is split between several destinations and the throughput is limited by the capabilities of the sending device and the size of the view. In case of multiple senders, more data could be put on the network, but it also raises the probability of collisions. The results of broadcast by unicast from both a single sender and simultaneous broadcasts by multiple senders can be seen in Figure 14. The throughput of broadcast by UDP with both small and large messages (1,000 and 30,000 bytes respectively) can be seen in Figure 15. As we can see, this throughput is considerably lower than that of unicast emulation. In addition to the lower overall throughput values, the network became extremely unstable when we were trying to stress it with broadcasts, which prevented us from gathering additional results of this type of broadcast. This probably results from the well documented tendency of WiFi network cards to switch to low bit rates (e.g., 2Mbps) when transmitting in broadcast mode.

7 Sample Applications Based on Chameleon

Chameleon was successfully used by several students’ academic projects as connectivity infrastructure in such applications as chatting, file sharing, media streaming and multiplayer games. See Figure 16 and Figure 17 for examples. In all cases, the students reported very positive experience and were able to focus on higher level aspects of the application rather than dealing with low level communication. Also, they were able to demonstrate cross platform portability and interoperability.
Figure 12: Reliability and resubmitted percentage under non-uniform load, short messages
Figure 13: Reliability and resubmitted percentage under non-uniform load, long messages
Figure 14: Throughput of broadcast simulated by unicast vs number of devices with single and multiple senders
8 Discussion

In this work, we have presented a new open-source, multi-platform, modular group communication system specifically targeted at smartphone devices, but not limited to them. We have described the main building blocks of the system, presented its design decisions, discussed their impact on application development and evaluated its performance. We have presented the tools supporting the development process and discussed porting of core application components to platforms supported by Chameleon through the same means as were used in the porting of Chameleon itself. Chameleon is currently available as an open source project in [2].

During the development process, we often faced technical difficulties caused by the unstable and heterogenous nature of the mobile devices world, incomplete documentation, sometimes only partial implementation of the specifications and hard to impossible emulation of connected devices. Additionally, on many devices, ad-hoc mode connectivity support does not come built-in and has to be enabled by external means such as [1]. It is our hope that using Chameleon will lift some of the concerns we faced from the shoulders of an application developer.

Our benchmarks have shown that the performance obtained when using Chameleon is comparable to what can be obtained using bare UDP, meaning that application developers can enjoy the benefits of Chameleon without sacrificing performance. The feedback that we obtained from students developing a host of applications over Chameleon was very positive.

8.1 Future work

Chameleon in its current state provides only the basic components needed for communication. It can benefit from additional support for high level features, such as publish-subscribe, multi-hop discovery mechanisms, authentication and encryption, as well as remote method invocation abstractions.

In addition, we plan to extend Chameleon by adding cross-compilation to additional platforms,
such as iOS by [35], supporting additional connectivity technologies such as NFC, Bluetooth, and cellular, and add an Internet wide discovery service. Once these are added, an interesting challenge would be to orchestrate all these technologies so that the system automatically communicates in the most suitable manner depending on the exact setting between each pair of devices.

References

Figure 17: P2P file sharing application between Android and J2ME devices


## A Examples

### A.1 XML headers

<table>
<thead>
<tr>
<th>XML file</th>
<th>XML code</th>
</tr>
</thead>
</table>
| **Message type** | <xml version="1.0">  
<xs:element name="MessageType">  
  <xs:complexType>  
    <xs:sequence>  
      <!-- 0 - Chat message -->  
      <!-- 1 - Control message -->  
      <xs:element name="type" type="xs:byte"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>  
</xs:schema> |
| **Chat message** | <xml version="1.0">  
<xs:element name="ChatMessage">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="room" type="xs:string"/>  
      <xs:element name="message" type="xs:string"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>  
</xs:schema> |
| **Control message** | <xml version="1.0">  
<xs:element name="ControlMessage">  
  <xs:complexType>  
    <xs:sequence>  
      <!-- 0 - Join room -->  
      <!-- 1 - Leave room -->  
      <xs:element name="action" type="xs:byte"/>  
      <xs:element name="username" type="xs:string"/>  
      <xs:element name="room" type="xs:string"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>  
</xs:schema> |

Notice that all three files have the same header and the same footer, with only the name of
the element changing. Apart from these mostly constant lines, the first file has only one line of
definitions and some comments. The two other files are not much more complex either. Currently,
Chameleon supports the following data types of the W3C Schema:

<table>
<thead>
<tr>
<th>W3C XML type</th>
<th>Value of maxOccurs flag</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:byte</td>
<td>None</td>
<td>byte</td>
</tr>
<tr>
<td>xs:int</td>
<td>None</td>
<td>int</td>
</tr>
<tr>
<td>xs:long</td>
<td>None</td>
<td>long</td>
</tr>
<tr>
<td>xs:double</td>
<td>None</td>
<td>double</td>
</tr>
<tr>
<td>xs:boolean</td>
<td>None</td>
<td>boolean</td>
</tr>
<tr>
<td>xs:string</td>
<td>None</td>
<td>String</td>
</tr>
<tr>
<td>xs:int</td>
<td>“unbounded”</td>
<td>Integer[]</td>
</tr>
<tr>
<td>xs:long</td>
<td>“unbounded”</td>
<td>Long[]</td>
</tr>
<tr>
<td>xs:string</td>
<td>“unbounded”</td>
<td>String[]</td>
</tr>
</tbody>
</table>

Additional types can be easily added by extending the serialization primitives of Chameleon.
### A.2 Messaging code of an application

<table>
<thead>
<tr>
<th>Action</th>
<th>Java code</th>
</tr>
</thead>
</table>
| **Sending a message**   | public void sendChatMessage(String room, String message) {
|                         |     Data data = new Data();                                                                                                                                                                |
|                         |     ChatMessage chatMsg = new ChatMessage();                                                                                                                                             |
|                         |     chatMsg.setRoom(room);                                                                                                                                                        |
|                         |     chatMsg.setMessage(message);                                                                                                    |
|                         |     data.integrateHeader(chatMsg, new ChatMessageSerializationAgent());                                                                                                           |
|                         |     MessageType mt = new MessageType();                                                                                                                                                |
|                         |     mt.setType(CHAT_MESSAGE);                                                                                                                      // Local constant |
|                         |     data.integrateHeader(mt, new MessageTypeSerializationAgent());                                                                                                           |
|                         |     sendData(data); // Implemented by the abstract level                                                                               |
| **Joining a chat room** | public void joinChatRoom(String room) {                                                                                                                                                |
|                         |     Data data = new Data();                                                                                                                                                                |
|                         |     ControlMessage ctrlMsg = new ControlMessage();                                                                                                                                  |
|                         |     ctrlMsg.setAction(ACTION_JOIN_ROOM);                                                                                                                                           // Local constant |
|                         |     ctrlMsg.setUsername(this.strUserName);                                                                                                                                          // Initialized by the app |
|                         |     ctrlMsg.setRoom(room);                                                                                                                                                           |
|                         |     data.integrateHeader(ctrlMsg, new ControlMessageSerializationAgent());                                                                                                          |
|                         |     MessageType mt = new MessageType();                                                                                                                                                |
|                         |     mt.setType(CONTROL_MESSAGE);                                                                                                               // Local constant |
|                         |     data.integrateHeader(mt, new MessageTypeSerializationAgent());                                                                                                           |
|                         |     sendData(data); // Implemented by the abstract level                                                                               |
| **Handling messages**   | // Overrides an empty abstract implementation                                                                                                                                            |
|                         | protected void receiveData(Member member, Data data, long messageId) {                                                                                                                     |
|                         |     MessageType mt = (MessageType) data.extractHeader(new MessageTypeSerializationAgent());                                                                                     |
|                         |     if (mt.getType() == CONTROLLER_MESSAGE) {                                                                                            |
|                         |         ControlMessage ctrlMsg = (ControlMessage) data.extractHeader(new ControlMessageSerializationAgent());                                                                 |
|                         |         // Continue with application specific handling                                                                                                                                         |
|                         |         handleControlMessage(member, ctrlMsg);                                                                                                                                           |
|                         |     } else if (mt.getType() == CHATMESSAGE) {                                                                                                                                          |
|                         |         ChatMessage chatMsg = (ChatMessage) data.extractHeader(new ChatMessageSerializationAgent());                                                                                |
|                         |         // Continue with application specific handling                                                                                                                                         |
|                         |         handleChatMessage(member, chatMsg);                                                                                                                                           |
|                         |     }                                                                                                                                                                                    |

---
The example above creates instances of serialization agent classes for the sake of clarity only. These classes are completely stateless and in a real application should be reused.

Notice how simple and readable becomes the code of an application once custom headers are built. In case of more complex messages that cannot be expressed by a single header file, but ones with logical dependencies between them, an application can easily implement its own serialization agent by combining those generated automatically and adding the interconnecting logic. This way, it will not have to descend to the level of primitives serialization and will still have the advantage of messages readable on multiple platforms. In fact, a simple form of headers’ conjunction can be seen even in this toy example. The MessageType header acts as a software port, allowing to decide whether the rest of the data has to be deserialized as a chat or as a control message.

B Testing

A non-negligible part of the development effort was devoted to testing the system. This testing was performed in two ways. One is the black box testing by developing applications and analyzing their behavior, such as those presented in section 7 and the benchmarking application whose results are presented in section 6.

The other is unit testing. All non-transport layers in Chameleon were thoroughly unit-tested. Because of the threading assumptions of the framework and its semi-asynchronous nature, such testing is not trivial and required development of a set of utilities, mocks and expressive, stackable event filtering predicates. When tested with these utilities, execution in a test environment resembles closely the execution in production, while the assertions are made on events available in the chain – their direction, type, order and values of their properties.

Each test begins with building a part of a stack such that the tested layer resides between two mock layers. Usually only one layer is placed between the mocks, but several could be placed as easily, as in figure Figure 18. All layers, including the mocks, are connected by an EventChain and an AsyncExecutor is assigned to them – as in production. Layers are initialized in the same way as in production and a view mock is passed to them, so tested layers cannot distinguish between
an execution of a test and normal operation.

The mocked layers only produce events when ordered to do so by a test and such events always travel in the direction of the tested section. Mocked layers consume all events received from the tested section and put them in an internal data structure. This structure supports various complex assertions. The assertions are performed separately on events received by the lower and the upper mock. Most assertions receive as parameter a filter predicate and each assertion is performed on the result of filtering the data structure with this predicate. Assertion operations can chose to modify the data structure, for example: count and remove all matching events. An internal event flushing logic guarantees that all asynchronous events, potentially being in transit in the event chain, are flushed before the assertion is made, thus ensuring consistent results on every execution.

Much of the assertion expressiveness depends on the expressiveness of filter predicates constructed by a test. Constructing such predicates, however, could be a rather complex task. Consider for example a predicate with a semantic of “an outgoing ACK message on two messages: \( m_i \) and \( m_j \) received from process P”. Such a predicate must:

- Examine the type of an event and verify that it is an OutgoingAddressedMessage
- Verify that the target of the message is process P
- Extract the outer header of the message and verify that it is marked as ACK
- Extract next header and verify that it contains IDs of messages \( m_i \) and \( m_j \) in any order

If coded explicitly, the size of the predicates code would usually by far surpass the size of the test using them. Especially considering that normally a test has to perform several assertions to test a behavior. Luckily, we can see that each part of the predicate logic depends only on the parts above it and can be easily reused. The testing framework contains a set of static factory classes – a factory for most types of events and a factory for each type of header used by the Chameleon. These factories produce simple filter predicates, which test only a small part of the needed logic, which are then combined to more complex and expressive ones by tests. Prevention of further code duplication in construction of the predicates is the responsibility of a test. For example, a class testing the AckNack layer contains a short utility method building the predicate testing that a message is outgoing ACK to messages \( m_1, m_2, \ldots, m_n \), which in turn is used by all interested test scenarios, keeping the individual tests short and readable. Some of the generic tests, such as validation that events are not related to the layer in test pass it freely are implemented on the abstract level, further shortening and focusing the testing code.