EPiC Series in Computing

Volume 56, 2018, Pages 11–22

Proceedings of the 5th International OMNeT++ Community Summit

Cross-layer Stack Design Framework in OMNeT++

Doganalp Ergenc[1] and Ertan Onur[2]

[1] Middle East Technical University, Turkey
doganalp.ergenc@ceng.metu.edu.tr
[2] Middle East Technical University, Turkey
eronur@metu.edu.tr

Abstract

While networking applications are getting more comprehensive, the information required to perform the algorithms running upon such applications is increasing. Even though the modular design of network stacks provides an important abstraction between layers, it is now necessary to use all layer-specific information in cooperation. Therefore, cross-layer applications are designed for years. However, implementing and testing them in network simulators are still complex. In this study, we implemented a cross-layer design onto TCP/IP stack and gave a guide to design this architecture in OMNeT++. We also investigated the design in an example use case that is a clustering algorithm for ad-hoc networks.

1 Introduction

Network applications have become more complex in design. They require various and precise information to optimize and improve existing designs and develop new applications that outgun the current state-of-the-art. Cross-layer architecture enables the designers to benefit from a variety of layer-specific information in a layered stack design. That is, it exploits the layered-structure of an architecture to promote inter-layer communication and procure flexibility to use layer-specific information through the whole (or between some specific) layers. In this sense, it is an architectural optimization technique that increases the control over the information flow. Cross-layer optimization is considered for a number of problems such as effective routing, clustering, energy conservation and caching [1][2][3].

Designing such a cross-layer structure for research problems of wireless networks is quite frequent. Therefore, researchers may need to implement a cross-layer architecture in simulation environments for their studies. However, alongside its advantages, it indeed has a development cost. Apart from learning how to use simulation tools, the design of more complicated architecture (i.e., cross-layer architecture) requires mastering the dynamics of a simulation tool. In this study, we discuss the fundamental steps to develop a cross-layer framework in OMNeT++ giving examples from our own design. We especially address the researchers who are not expert in programming or OMNeT++ structure, and also other developers to share our development model. Our main contributions in this study are:

A. Förster, A. Udugama, A. Virdis and G. Nardini (eds.), OMNeT 2018 (EPiC Series in Computing, vol. 56), pp. 11–22

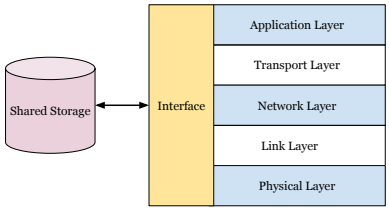
- We review three well-known cross-layer architectures, and some example OMNeT++ implementations for cross-layer designs (Section 2).
- We give a detailed guideline of a cross-layer stack architecture in OMNeT++ (Section 3). This is a complete roadmap with sample codes to create cross-layer TCP/IP stack. We also publish a ready-to-use implementation for a generalized cross-layer design[1]. This implementation is prepared for OMNeT++ v5.1.1 using *inet* v3.6.0 framework.
- We present the full picture of OMNeT++ modules to illustrate an example cross-layer design in our use case (Section 4).

2 Related Work

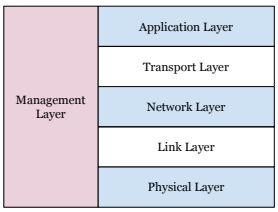
Designs with different requirements determine the structural aspects of the cross-layer architecture. For instance, frequency, quantity, and direction of information sharing between layers are some of those requirements. Generally, three main cross-layer architectures are proposed in the literature; (a) shared storage, (b) management layer and (c) direct connection [4]. In a cross-layer design, (a) a shared storage is accessed by the layers to extract and update commonly-used information. It could be considered as a micro-level database with layer interfaces as shown in Figure 1a. Especially when all layers need to share information, defining a single shared storage for common usage is much more effective than the one-to-one connection between each layer [1]. (b) Figure 1b shows the management layer that is (vertically) placed as a proxy layer between multiple layers and has a role to manage different parameters that belong to other layers. It is able to make asynchronous requests to other layers to obtain or update parameters. Similarly, it can be accessed by any other layer for application-specific purposes [2]. (c) For the architecture where fewer layers need to work in cooperation, connecting them directly could be the least-cost option. Direct connection in cross-layer schema means that related layers are connected without any intermediary mechanism as it exists in the shared storage and the management layer architectures [3][5]. Figure 1 illustrates all those cross-layer architectures.

There are also some studies that promote the implementation of such approaches in OMNeT++. Massin *et al.* [6] propose a cross-layer design for content transmission. They use a cross-layer architecture with a management layer called XLI. However, they focus on the physical layer and link layer designs instead of a cross-layer architecture to optimize the quality of service (QoS) for multimedia applications. Even though implementation-specific details of radio access and resource allocation are well-explained in the study, it is not clear enough to be referenced for designing a cross-layer stack. Mohaghegh *et al.* [7] focus on QoS in his implementation to decrease delay in packet processing. They analyze the average processing time of the packets with different priority and compare plain and cross-layer architectures. It is stated that the experiments are conducted in OMNeT++ creating a sensor network but the implementation details are not presented in the paper. In [8], a cross-layer protocol for wake-up radios, DoRa, is presented. An abstract structure of OMNeT++ design is given in the paper, and also performance results for the new wake-up radio mechanism are presented. However, the cross-layer design in this paper offers a limited collaboration between the physical layer and link layer. Since such limited cross-layer design satisfies the requirements of the protocol, there is no discussion for the implementation details of a generic cross-layer architecture. Lastly, [9] presents an XML-based structure to pass cross-layer information through control info, which is a built-in method in OMNeT++. It does not provide a complete cross-layer design but an

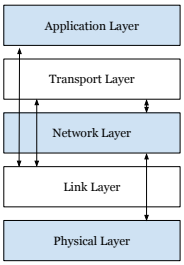
[1]The framework is published on GitHub at **github.com/d0d0d0/omnet-crosslayer**



(a) Shared storage



(b) Management layer



(c) Direct connection

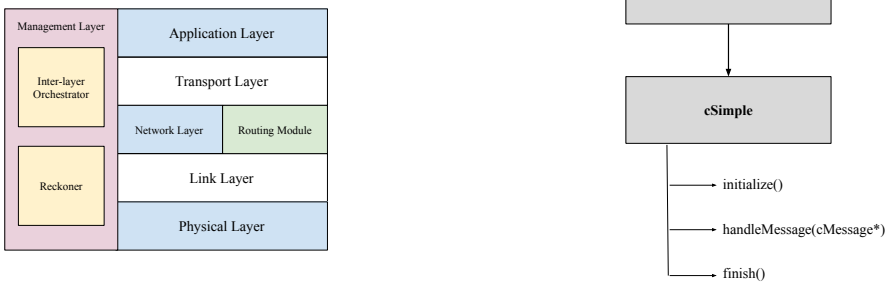
Figure 1: Three main cross-layer architectures: shared storage, management layer and direct connection.

insight to improve current packet-passing mechanism. None of those studies presents a guideline to design a generic cross-layer architecture but discusses the results of their proposals in specific subjects. Therefore, this study addresses the absence of the details for such a generic framework.

3 Implementation of Cross-layer Framework

In this section, we present the implementation details of our cross-layer framework. We employ the management layer approach shown in Figure 1b. The main reason for choosing this approach is its convenience for generalization with a reasonable development cost. For instance, integrating a shared storage or management layer to the layered stack incurs almost the same development cost: (a) defining a new module as a component and (b) connecting this component to all other layers. However, while a shared storage depends on a passive component used by other layers, a management layer can reactively orchestrate other layers and also can be used as passive storage. Another alternative, direct connection of layers, is effective only when a few numbers of layers need to communicate with each other. Because OMNeT++ has built-in features such as embedding *control info* and emitting *signals* to satisfy this kind

of limited inter-layer connection. However, its extensibility is limited when all layers require inter-communication. Therefore, we implemented the management layer to design a general-purpose framework onto the 5-layer stack where each layer can potentially share information with others.



(a) The management layer is placed vertically to other layers. While inter-layer orchestrator manages the communication between any two layers, reckoner implements the specific logic of the related protocol or design.

(b) OMNeT++ extension for the management layer

Figure 2: Abstract representations of the management layer and the overall stack.

The management layer in a cross-layer design has multiple roles:

- Collecting fresh information/parameters from other layers,
- Evaluating the information obtained from other layers simultaneously to make decisions,
- Forwarding any information from one layer to another to orchestrate collaboration.

Figure 2a shows the architecture having such roles. The management layer is vertically connected to the stack to be in communication with all layers. It basically consists of two submodules, the inter-layer orchestrator and the reckoner. The inter-layer orchestrator forwards the packets for communication between layers. On the other hand, the reckoner uses related packets coming from other layers to perform protocol-specific logic. In this sense, it is the brain of a cross-layer stack with the management layer.

There are three main steps to implement this architecture in OMNeT++:

1. **Definition of the management layer** includes the implementation of substeps for creating a new module that is able to communicate with other layers and make protocol-specific operations.

2. **Extension of other layers** shows how to extend related layers to work in collaboration, and also connect them to the management layer. This step is totally independent of the actual working dynamics of related layer. That is, the extension of a layer does not require to know internal details of that layer. Therefore, it is a generalized process.

3. **Creation of a new node** is the final step to compound the management layer and other extended layers in a single structure to constitute a complete cross-layer stack.

In the rest of this section, the overall design will be explained considering those steps.

3.1 Definition of the Management Layer

The fundamentals for creating the layer are given here. For the sake of simplicity, the cross-layer structure, which is formed by following the steps, does not contain the reckoner. Because the outcome of the steps is a generalized cross-layer framework and does not embrace a protocol-specific logic. One can add the reckoner submodule after implementing the backbone structure that is exemplified through this section. Moreover, an example use of the reckoner is presented in Section 4.

Definition of the management layer consists of two steps. First, we create a NED file that includes required gates to communicate with other layers and also specific parameters for the management layer. Then, we define and implement the gates in C++ classes and handle the inter-layer communication.

3.1.1 Layer-specific parameters and gates in NED files

Gates are basically full- or half-duplex channels to send interlayer packets. In this design, we used half-duplex gates that are specifically defined as input or output gate. *Parameters*, on the other hand, include different types of module-specific variables and signals that are detailedly explained in the OMNeT++ manual. Figure 3 shows the example gate definition for a 5-layer stack. There exist an input and output gate for each layer in order. Those gates are needed to be defined in C++ to be able to process incoming packets, and also to send packets via output gates to other layers.

```
simple ManagementLayer
{
    parameters:
        @display("i=block/buffer");

    gates:
        input appIn;
        output appOut;

        input transIn;
        output transOut;

        input networkIn;
        output networkOut;

        input linkIn[];
        output linkOut[];

        input phyIn[];
        output phyOut[];
}
```

Figure 3: NED for the management layer.

Note that, the example in Figure 3 is not the only way to implement such a structure. For instance, one can define full-duplex gates that handle both incoming and outgoing packets, and it naturally decreases the number of gates to manage. Even though such an implementation is leaner for experienced programmers, its management (i.e., handling incoming packets and sending outgoing packets through a single gate) is harder to practice. Therefore, we used two separate gate definition to make the implementation (i.e., roles of the gates) comprehensible.

Through all study, the motivation for this kind of implementation-level preferences is keeping sample codes as simple as possible.

3.1.2 Packet-handling scheme for inter-layer communication in C++

To program the main logic of the layer, we need to define C++ classes in parallel with NED file. OMNeT++ has extendable modules having the functions to (a) make initial definitions, (b) handle incoming packets and (c) handle module termination. That is, those functions are ready for extension/reimplementation with built-in event handlers. *cSimpleModule* class gives require interfaces to extend and implement our custom management layer module. Besides, basic send/receive functions are also implemented in that module. Figure 2b shows the simple extension of related modules and functions for general use.

Definition of the new module and porting gates in NED to C++ objects are very trivial. Figure 4a and 4b show the definition and implementation of those steps in the *initialization* (*cSimpleModule::initialize()*) function of the base module. They are actually not obligatory; it is also possible to access to gates through their names as strings. It is syntactically possible, albeit not a convenient and clear use in practice. Both uses of them are exemplified in Figure 6b but porting them to C++ variables in use is recommended.

```
class ManagementLayer : public cSimpleModule
{
    protected:
        cGate *appInGate = nullptr;
        cGate *appOutGate = nullptr;

        cGate *transInGate = nullptr;
        cGate *transOutGate = nullptr;

        cGate *networkInGate = nullptr;
        cGate *networkOutGate = nullptr;

        virtual void initialize() override;
        virtual void handleMessage(cMessage*) override;
        virtual void finish() override;
};
```

(a) Header file and definitions

```
void ManagementLayer::initialize()
{
    appInGate = gate("appIn");
    appOutGate = gate("appOut");
    transInGate = gate("transIn");
    transOutGate = gate("transOut");
    networkInGate = gate("networkIn");
    networkOutGate = gate("networkOut");
};
```

(b) Initilization phase in C++

Figure 4: The Initialization for gates in C++.

After the gates are defined, handling incoming packets and sending outgoing packets via related gates are trivial. Basically, *message handling* function of the base module is triggered whenever a packet arrives in a module. Besides, it is possible to take action with respect to the input gate that a packet has come. Figure 5a shows the backbone of packet handling function that is controlled by simple conditionals. Note that, what to perform in such conditionals completely depends on the protocol-design. Similarly, Figure 5b shows a brief use of *send* (*cSimpleModule::send(cMessage*, cGate*)*) function of the base module.

An important point about inter-layer communication is worth discussing here, the packet-passing mechanism. We prefer to pass custom packets that are defined for communication between the management layer and other layers through the gates. OMNeT++ has its own message definition interfaces that are defined in *.msg* files. They can be easily linked to C++ objects, to be more precise, C++ objects can be used as the content of the messages. For instance, *CrossTransMsg* is a custom message to perform cross-layer communication between the management layer and transport layer in Figure 5b and it may contain transport layer-

```
void ManagementLayer::handleMessage(cMessage *msg)
{
    if(msg->getArrivalGate() == appInGate){
        //Take action for incoming packets from Layer 5
    } else if(msg->getArrivalGate() == transInGate) {
        //Take action for incoming packets from Layer 4
    } else if(msg->getArrivalGate() == networkInGate) {
        //Take action for incoming packets from Layer 3
    } else if(msg->getArrivalGate()->isName("linkIn")) {
        //Take action for incoming packets from Layer 2
    } else if(msg->getArrivalGate()->isName("phyIn")) {
        //Take action for incoming packets from Layer 1
    }
};
```

(a) Handling packets coming from different layers

```
void ManagementLayer::sendTransLayer(int type)
{
    CrossTransMsg *packet = new CrossTransMsg("CrossTransMsg");
    packet->setType(type);
    send(packet, transOutGate);
};
```

(b) Sending a packet through defined gate

Figure 5: Controlling the inter-layer communication.

related information represented by C++ objects. Apart from this method, OMNeT++ has two main built-in features to perform such communication, (a) signaling and (b) control info. (a) Signaling is basically a publish-subscribe method. A signal is actually a value that can be defined as some primitive data types such as integer and float, and also object pointers. When a module emits (or publishes) a signal, all other modules that are subscribed to that signal can access its value. It is the main method to collect statistics in OMNeT++. For instance, when each module signals the number of packets it has processed, the host of those modules obtains the statistics for the total number of packets processed by subscribing to the signals. For such purpose, signaling is quite practical: defining signals in NED file for source modules and registering to them in receiver modules are enough. (b) Control info, on the other hand, is a ready-to-use object and can be attached to other message objects. For instance, one can attach a control info to a MAC frame in the link layer to be detached and processed in the application layer.

Those packet-passing mechanisms have their own advantages and disadvantages. While signaling is perfect to collect statistics using primitive data types, sharing more complex data types via signals and processing them during network lifetime are more challenging. One needs to implement related listener classes extending *cListener* and overwriting its methods to handle custom signals. Even if control info requires almost no extra implementation overhead (except the definition of a custom object to pass related parameters), is not a convenient way if it is required to gather a collective information from multiple modules to process them simultaneously. Our proposed scheme, on the other hand, requires extra gate definitions. For the stack design, our scheme is like a natural extension of current TCP/IP stack implementation whose inter-layer packet-passing mechanism is also performed through gates. Therefore, we believe that it is easy to adopt and implement.

3.2 Extension of Other Layers

Extension of other layers is a similar process that is explained for the definition of the management layer. That is, they all need to (a) have input/output gates to communicate with the management layer, (b) extension in message-handler functions for interlayer communication. Other layer-specific functions need to be implemented and integrated to message handling function properly. Thanks to the modular and extendable implementation of the OMNeT++ modules, the methodology for implementing the layers is quite trivial and each of them requires nearly the same process. For instance, Figure 6a shows the extension of the link layer which is

IdealMac in OMNeT++ to create a *CrossIdealMac*. Modification/extension in NED file, definition and initialization of the gates, and the extension of the message handling and finishing functions are shown in the figure as well.

```
package src.LinkLayer;

import inet.linklayer.ideal.IdealMac;

module CrossIdealMac extends IdealMac
{
    parameters:
        @class(CrossIdealMac);

        @signal[notForUsSignal](type="long");
    gates:
        input crossIn @labels(CrossControlInfo/down);
        output crossOut @labels(CrossControlInfo/up);
}
```

(a) NED file extension

```
void CrossIdealMac::initialize(int stage)
{
    IdealMac::initialize(stage);

    if (stage == INITSTAGE_LOCAL) {
        crossInGate = gate("crossIn");
        crossOutGate = gate("crossOut");
    }

    isRedundant = checkRedundancy();

    notForUsSignal = registerSignal("notForUsSignal");

    CrossSelfMsg* packet = new CrossSelfMsg();
    packet->setM_type(inet::LINK_UPDATE_ENERGY);
    scheduleAt(simTime() + UPDATE_ENERGY_PERIOD, packet);
};

void CrossIdealMac::handleMessage(cMessage *msg)
{
    if (msg->getArrivalGate() == crossInGate) {
        handleCrossLayerMessage(msg);
    } else {
        if(msg->getArrivalGateId() == lowerLayerInGateId) {
            sendRSSI(msg);
        }
        IdealMac::handleMessage(msg);
    }
};
```

(b) Extension of the initialization and message handling functions

Figure 6: Extension of layers modifying NED and C++ file.

Note that, the example code pieces in Figure 6a are just simplified versions; they are illustrative for the fundamentals of the design taken from the use-case presented in Section 4. There are many other implementation-specific details for defining modules, handling namespaces etc. that are out of the scope of this study.

3.3 Creation of a New Node

After the definition of the management layer and the extension of other layers, all those layers need to be connected to each other under a single node definition. This, at least for this example, does not require any reimplementation in C++ but a structure in a NED file. OMNeT++ already has a variety of node definitions which have different capabilities. For example, a generic host for wireless communication, more specific ad-hoc host as an extension of generic wireless host and routing hosts implementing different routing algorithms exist in the *inet* framework. Therefore, instead of defining a node from scratch, modifying one of them depending on one's projects requirements would be easier and faster.

Figure 7 shows how all gate definitions in extended layers are gathered and connected in a single node definition. Note that, it is not always possible to directly use those gates in the node definition; one needs to consider the built-in relationship between modules. For example, the use of the physical layer and link layer are directly integrated. Therefore, it requires a transition step with extra gate connections before connecting them in the node definition. In the figure, a similar scenario exists and thus there are differences between the original gate names defined in previous figures and Figure 7.

```
module AdhocNode extends WirelessHost
{
    parameters:
        forwarding = default(true);
        string crossType = default("LowestIDClustering");

    submodules:
        cross: <crossType> like ICrossLayer {
            @display("p=527,287");
        }

    connections allowunconnected:
        cross.appOut --> udpApp[0].crossIn;
        udpApp[0].crossOut --> cross.appIn;

        cross.transOut --> udp.crossIn;
        udp.crossOut --> cross.transIn;

        cross.networkOut --> networkLayer.crossIn;
        networkLayer.crossOut --> cross.networkIn;

        for i=0..sizeof(radioIn)-1 {
            cross.linkOut++ --> wlan[i].XmacIn;
            wlan[i].XmacOut --> cross.linkIn++;

            cross.phyOut++ --> wlan[i].XphyIn;
            wlan[i].XphyOut --> cross.phyIn++;
        }
}
```

Figure 7: NED file for the definition of a new node extending wireless host.

Another important point is the selection of extended modules for an active use in our node definition. That is, the node needs to be configured to include our extended modules. It is, for example, handled in the base node definitions (*NodeBase* and *StandartHost*) which are extended by the OMNeT++ module, *WirelessHost*. This is quite trivial and parametric but still a step that requires attention.

4 A Use Case: A Cross-layer Clustering Algorithm for Ad-hoc Networks

There is a variety of use cases for cross-layer designs in wireless networks [4][10]. Security, QoS, and mobility are the major issues to handle using cross-layer architectures. In this section, our own cross-layer design to implement a clustering algorithm for mobile ad-hoc networks is presented in a use case study. Note that, rather than internal details and analytical discussion of the study, we only investigate how we design and use cross-layer architecture for clustering. That study has been recently published [11], and all empirical results are collected using the framework that we presented in this paper.

Briefly, clustering in ad-hoc networks is a well-known technique to be able to manage large-scale networks. Even though its uses cases are limited for the networks that heavily depend on central controller mechanisms and predeployed infrastructures, the technique is quite important for ad-hoc networks and sensor networks that are more self-organized and distributedly managed [12]. Clustering consists in grouping the nodes and assigning them special roles, like cluster head and gateways. Cluster heads are the leaders that are responsible for their local neighborhood and have different roles such as resource allocation, find routes and handle handovers in mobile networks [13]. Selecting those leaders is the fundamental challenge in clustering algorithms and requires the information originated from different stack layers. At this point, a cross-layer

design becomes crucial to be able to collect and evaluate such information. In this use case, we present the general structure of Density-aware Probabilistic Clustering Algorithm (PCA), that uses individual node degrees as an indicator of the probability of being cluster head.

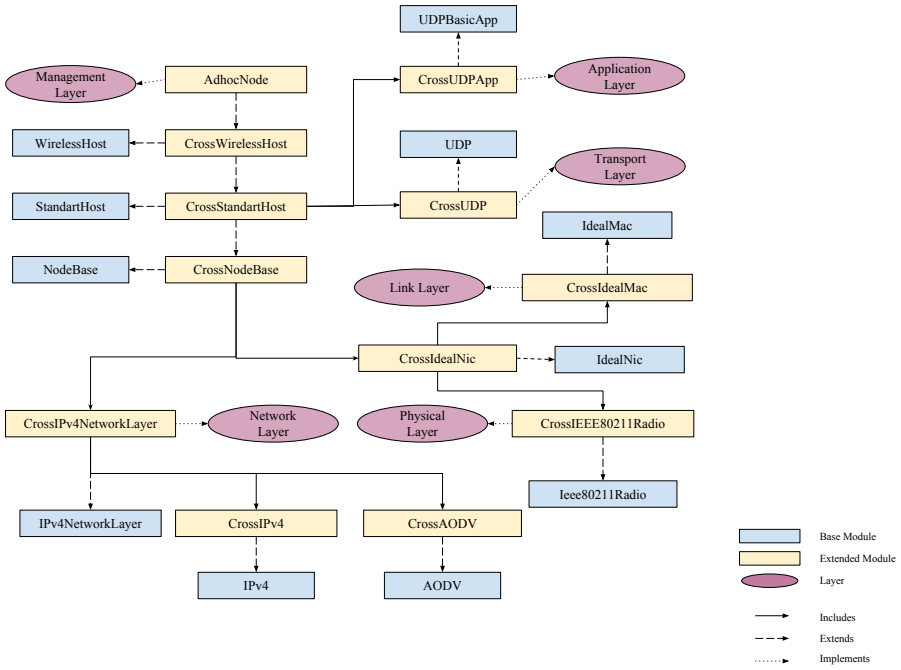


Figure 8: The overall OMNeT++ architecture for PCA that shows all module extensions.

Figure 8 shows the overall OMNeT++ architecture for cross-layer design in PCA. Note that, even though physical layer is not directly involved in cross-layer structure, it is also extended to form a complete framework. Depending on the design requirements, one does not have to cover all layers vertically. For PCA, only four layers (i.e., 3+1 including management layer) are necessary. (1) For the link layer, *CrossIdealMac*, is designed as an extension of *IdealMac*. In this module, received MAC packets are forwarded to the management layer where the reckoner counts the number of neighbors in 1-hop to compute the probability of being cluster head for the receiver node during the bootstrapping phase. In this phase, nodes explore their local neighborhood to understand if they are eligible to be a cluster head. In this method, a higher number of neighbors increase the probability for being cluster head. At the end of bootstrapping phase, some of the nodes claim themselves as cluster head depending on the probability calculated by reckoner in the management layer. Then, they need to announce such claim and the maintenance phase starts. For the announcement, (2) we implemented *CrossUDPApp* extending *UDPBasicApp* for the application layer to share control packets periodically. Each node broadcasts control packets containing its identifier and role, i.e., whether it has claimed itself as a cluster head with a UDP packet. This module also forwards received control packets to the management layer so that the reckoner can analyze the information coming from neighbor nodes and realize if there are any clusters in the node's neighborhood. If there are multiple clusters, the reckoner chooses to join the cluster with the minimum identifier that is

the identifier of the cluster head. Eventually, every node resides in a cluster as a cluster head or ordinary node. After the reckoner made this decision, (3) the management layer informs the network layer about the cluster head. The network layer, *CrossIPv4NetworkLayer*, is implemented as an extension of *IPv4NetworkLayer*. Besides, *CrossAODV* is our routing module and it is integrated into the network layer. Once the network layer becomes aware of the identifier of the cluster head node, the data packets for end-to-end communication are started to forward to that node by the network layer. In this sense, *CrossIPv4NetworkLayer* is configured to communicate with related cluster head so that it can orchestrate the communication in a cluster. During network lifetime, the management layer constantly informs *CrossIPv4NetworkLayer* in case of changing cluster. Figure 9 shows the flow chart for PCA including bootstrapping and maintenance phases and inter-layer communication.

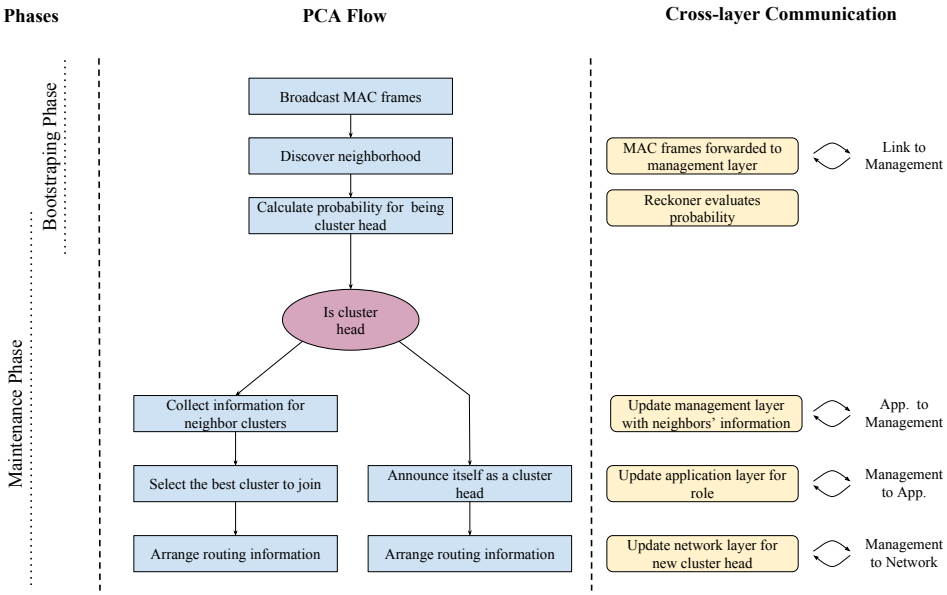


Figure 9: Flow chart for PCA shows cross-layer functionalities. On the left side, the phases of PCA are associated with related flow-tasks. On the right side, cross-layer communication steps are associated with those tasks as well. Besides, the direction of cross-layer communication is noted for each step.

The performance of the algorithm evaluated using this model and extensive results are presented in [11]. This is a very concrete example of a cross-layer design in OMNeT++ to test a design in different scenarios such as different channel models, mobility models, and network density. The whole model is designed using the guidance in the scope of this study, and most of the code pieces shown here are the simplified versions of the original model.

5 Conclusion and Future Work

The main focus of this study is to provide a complete, understandable and practical guide to design cross-layer architecture for OMNeT++. We presented our cross-layer framework and showed a concrete system design using such a cross-layer approach to test and simulate research

projects.

There are indeed different methods to develop a cross-layer architecture and they are also required to be examined and compared to find the one (a) with a shallow learning curve, (b) with less development cost and (c) is adaptable to a variety of projects. Besides, the implementation-specific issues like packet-passing mechanisms are needed to be evaluated empirically to understand performance overhead. Such optimization issues are left as the future work after this study.

6 Acknowledgement

This work is partially supported by ASELSAN Inc.

References

[1] R. Glitho, C. Fu, and F. Khendek. Cross-Layer Design for Optimizing the Performance of Clusters-Based Application Layer Schemes in Mobile Ad Hoc Networks. In *Proc. of the 4th IEEE Consumer Communications and Networking Conference*, pages 239–243, Jan 2007.

[2] M. K. Denko, J. Tian, T. K. R. Nkwe, and M. S. Obaidat. Cluster-Based Cross-Layer Design for Cooperative Caching in Mobile Ad Hoc Networks. *IEEE Systems Journal*, 3(4):499–508, Dec 2009.

[3] R. Mehta and D. K. Lobiyal. Energy efficient cross-layer design in MANETs. In *Proc. of the 4th International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 448–453, Feb 2017.

[4] V. Srivastava and M. Motani. Cross-layer design: a survey and the road ahead. *IEEE Communications Magazine*, 43(12):112–119, 2005.

[5] J. Peng, H. Niu, W. Huang, X. Yin, and Y. Jiang. Cross layer design and optimization for multihop ad hoc networks. In *Proc. of the IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pages 1678–1682, March 2017.

[6] R. Massin, C. Lamy-Bergot, C. J. Le Martret, and R. Fracchia. OMNeT++-Based Cross-Layer Simulator for Content Transmission over Wireless Ad Hoc Networks. *EURASIP Journal on Wireless Communications and Networking*, 2010(1):502549, Jan 2010.

[7] M. Mohaghegh, C. Manford, and A. Sarrafzadeh. Cross-layer optimisation for quality of service support in wireless sensor networks. In *Proc. of the IEEE 3rd International Conference on Communication Software and Networks*, pages 528–533, May 2011.

[8] Jean Lebreton and Nour Murad. Implementation of a Wake-up Radio Cross-Layer Protocol in OMNeT++, MiXiM. *CoRR*, abs/1509.03553, 2015.

[9] Laura Marie Feeney. Managing cross layer information in OMNeT++ network simulations.

[10] B. Fu, Y. Xiao, H. J. Deng, and H. Zeng. A Survey of Cross-Layer Designs in Wireless Networks. *IEEE Communications Surveys Tutorials*, 16(1):110–126, First 2014.

[11] Doganalp Ergenc, Levent Eksert, and Ertan Onur. Density-aware Probabilistic Clustering in Ad hoc Networks. In *Proc. of the IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, Batumi, Georgia, jun 2018.

[12] O. Boyinbode, H. Le, A. Mbogho, M. Takizawa, and R. Poliah. A Survey on Clustering Algorithms for Wireless Sensor Networks. In *Proc. of the 13th International Conference on Network-Based Information Systems*, pages 358–364, Sept 2010.

[13] Craig Cooper, Daniel Franklin, Montserrat Ros, Farzad Safaei, and Mehran Abolhasan. A comparative survey of VANET clustering techniques. *IEEE Communications Surveys & Tutorials*, 19(1):657–681, 2017.