Membrane System as a Communication Interface between IoT Devices

Šárka Vavrečková[1]

[1]Silesian University in Opava, Bezručovo nám. 13, Opava, Czech Republic

Abstract

Membrane systems can be used to describe transfer of objects between different locations (membranes) and their eventual transformation. Network transmission protocols provide something similar, especially the transfer of data. In this paper, we describe the use of a membrane system for modeling data transmission between IoT devices, while it is possible to take this model as a generalization of the transmission operation transferable to other protocols or to various programming languages. The paper also discusses the possibility of using rules of the membrane system to create a simple firewall.

Keywords

membrane systems, internet of things, protocol, membrane firewall

1. Introduction

Membrane computing is a framework of parallel distributed processing introduced by Gheorghe Păun in 1998. Information about this paradigm is available in [1, 2, 3], or the bibliography at http://ppage.psystems.eu/ [2022-06-08]. Membrane systems are based on the hierarchical structure of membranes in cells and can be used to model distributed computing. Mathematical models of membrane systems have been called P Systems.

Objects located in membranes pass between membranes according to defined rules, similar to how various substances are transferred between membranes in a biological cell. The system usually works in parallel (depending on the selected mode).

The Internet of Things (IoT) is a term that is very difficult to define. IoT devices are mostly small, inconspicuous devices with simple functionality and low consumption, whose strength lies primarily in their interconnection. We talk about smart devices, but the smartness is in their interconnection and usage of the data obtained from these devices.

IoT devices can be simple sensors detecting e.g. the temperature, motion, humidity, light, or alarms, mechanisms for handling lighting, opening or closing windows, or passive recipients of data (e.g. displays). It can be more complex devices with multiple functionalities, communication points, gateways to other network types, or common devices such as smartphones. We encounter IoT devices at home, in companies, hospitals, industry, agriculture, on streets,...

In principle, IoT devices are computationally independent of each other and work in parallel. Parallel processing can also be described using membrane systems, where the data transfer between devices can be modeled using evolution rules.

The use of membranes or similar principles in the IoT world has been considered for years. Villari et al. in [4] introduce the concept of "osmotic computing" as a paradigm, the main purpose of which is to increase the accessibility of resources and services in a computer network (e.g. IoT network), including cloud services. The authors present the concept of micro-services gradually migrating from the cloud (physically in large data centers) to the edge of the network (edge computing), i.e. they are performed on devices in the internal network. The paradigm is motivated by procedures from biology or chemistry, where solvent molecules pass through a semi-permeable membrane into other regions in the environment with higher solute concentration (osmosis).

The issue is further developed by the paper [5], which considers the way in which micro-services, in particular, can migrate between the cloud and edge resources and focuses more on the Internet of Things. The authors of [6] deploy micro-services in a hospital application. Datta and Bonnet in [7] show the use of MELs in securing connected "smart" cars and other similar devices.

We follow up on paper [8], in which the concept of using the P system as an interface to IoT devices is presented. Here we slightly modify the concept, add new types of objects corresponding to various kinds of messages in the IoT system, and strictly separate the different functionalities of the system. We also discuss the possibility of implementing a simple firewall at the membrane system level.

ITAT'22: Information technologies – Applications and Theory, September 23–27, 2022, Zuberec, Slovakia

sarka.vavreckova@fpf.slu.cz (Š. Vavrečková)

2. Preliminaries

2.1. Membrane Systems

We assume the reader to be familiar with the basics of formal language theory and membrane computing. For further details, we refer to [9] and [3].

As mentioned above, the basis of membrane systems is a membrane structure. A membrane can contain objects and/or nested membranes. The main membrane contains all the other membranes, we call it the "skin membrane". Objects can be handled using evolution rules. Figure 2 shows a membrane system with one skin membrane and four nested membranes. Almost all membranes contain at least one object (objects can be usually simply denoted by letters, but here we use more complex objects with various parameters).

Definition 1 ([2], [10]). *Let H be a set of labels. A P System of a degree m, $m \geq 1$, is a construct*

$$\Pi = (V, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m)$$

where:

(i) *V is a nonempty alphabet, its elements are called objects,*

(ii) *μ is a membrane structure consisting of m membranes, the membranes are labeled by the elements of H,*

(iii) *w_i, $1 \leq i \leq m$, are strings representing multisets over V associated with the region of the i-th membrane in μ,*

(iv) *R_i, $1 \leq i \leq m$, are finite sets of evolution rules associated with the region of the i-th membrane in μ; an evolution rule is a pair (u, v), also written $u \to v$, where*

- *u is a string over V,*
- *$v = v'$ or $v = v'\delta$, where v' is a string over $\{a_{here}, a_{out}, a_{in_j} \mid a \in V, 1 \leq j \leq m\}$, and δ is a special symbol $\notin V$ representing dissolution of membrane.*

The objects can be transported by the evolution rules through membranes due to the targets out *(to the parental membrane) or* in *(to the child membrane specified by the index), or they remain in the original membrane (*here*).*

Details and examples can be found in [2] and [10].

2.2. Internet of Things

We can find a lot of definitions of the Internet of Things, but none of them is fully descriptive. Their formulation depends on the usage of such specific IoT structure. In [11] there are several definitions taken from multiple sources. We can compose the following definition from them:

Definition 2 ([11]). *The Internet of Things (IoT) is a network of various types of smart objects (so called things) and devices. The things are connected to the Internet and communicate with each other with minimum human interface. They are embedded with abilities as sensing, analyzing, processing and self-management based on interoperable communication protocols and specific criteria. These smart things should have unique identities and personalities.*

There are several common communication models (or communication patterns) used in IoT networks, the model Publisher-Subscriber is quite common because it is closer to the needs of IoT than other models. There are three types of components: publishers produce data, subscribers consume and process data, and the controller (often called broker or server, depending on the specific protocol) as the central point of the network mediates data. Publishers send data to the controller, not to subscribers, they don't need to know about the existence of subscribers. Each node with the role of a subscriber subscribes to particular types of data (often called topics) to the controller, and the controller sends the requested data to all the subscribers interested in them. Sensors are examples of publishers (producers); actuators (simple motors) or displays are examples of subscribers (consumers).

In practice, various higher-level protocols are used in IoT networks, such as MQTT, XMPP, CoAP, AMQP, or simply HTTP as in classic computer networks. Furthermore, we will mainly follow the MQTT protocol, in terms of terminology, message types and communication scheme.

More details about IoT network communication models, including protocols, can be found in [12]. Very clear and brief introduction to MQTT messages is in [13].

3. Model of System

The given definition of the P System does not meet our requirements: we need a slightly more dynamic structure, where it is possible to create new objects due to external influences (e.g. to generate an object containing temperature data), to pass objects to a real device, but also to generate or delete rules. There are two possibilities:

1. We can modify the definition.
2. We can add an additional layer above the membrane structure, this layer will carry out the stated tasks.

The second option is more feasible for our purposes, however, with the possibility to change the set of rules in membranes from outside (there is no need to change the definition of membrane system). The set of rules will be changed continuously, but the impact will only be a change in the operation of the system, not collisions or system errors.

The added control layer will be able to affect objects and rules at each step of the computation. Figure 1 demonstrates the whole structure. The control layer is above the membrane structure, this layer communicates using a suitable protocol with other components: a local network and, indirectly, the Internet, the cloud.

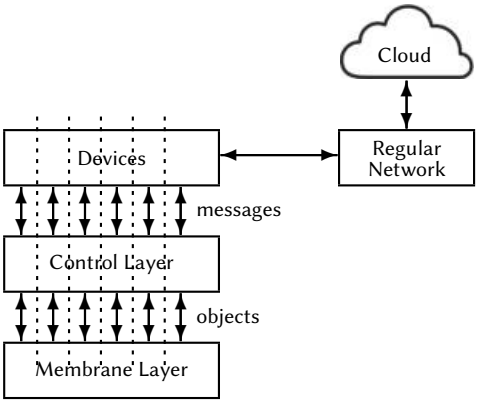

Figure 1: Communication Architecture

The membrane layer contains a P System with a membrane structure, objects and rules as prescribed by the definition. The only change from the definition is the addition of semantics. The objects contain semantic information (e.g. the identifier of the sending membrane, the published data, the credentials when connecting). The rules only take this information into account in their notation; the semantic information is not changed by rules, only transited, or a new object with semantic information is created.

The control layer keeps the state of the P System (references to membranes and rules) and other information – topics, subscriptions etc. And it makes interventions to the membrane layer (adding new objects, picking up some other objects etc.).

Each device communicates with one assigned component from the control layer, and each component is connected to one assigned membrane from the membrane layer. The devices do not communicate directly with each other, nor do the components, only the membranes forward objects to each other.

The separation of the data transfer itself into the membrane layer has another positive effect: it is not necessary for all devices to use the same protocol and provide data with the same meta-information, the control layer can perform reconciliation.

3.1. Membrane Layer

Figure 2 shows the membrane structure of an example IoT system with one controller and several components.

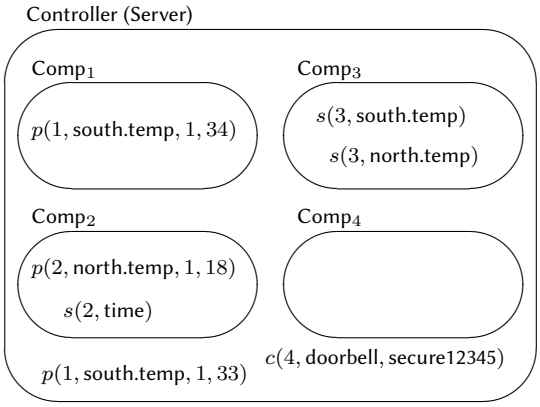

Figure 2: Example of IoT Membrane Structure

The components Comp_1 and Comp_2 are thermometers, the first one on the south side of a house and the second one on the north side of a house (see the names of the topics to which they contribute). Both thermometers generate temperature data at regular intervals, which means that the object p appears regularly in the corresponding membranes. In the next step, this object is transferred to the outer membrane using a rule.

The length of the interval is set on the given device, it can be different on each device. In our case, the object for the first component is generated in each step of the membrane system operation; the second component has a bit longer interval.

Comp_2 is a more sophisticated device that can also display time, it sent an order to synchronize the time data (however, there is no component in the system yet that would serve as a publisher for the given topic).

Comp_3 is a display that has just been activated and it is sending an order for data of the first two components (temperature topics).

The fourth component is being activated – an object c has appeared in the skin membrane environment for connecting this membrane to the system (activating the corresponding device). According to the semantic data of the object, it may be a doorbell.

In Figure 2 we can see several objects in various membranes. Each of the objects also has properties. The objects passing through membranes represent messages sent between components. We use the following types of objects in our system:

- $c(\text{ID, credentials})$ is sent to negotiate a connection to the controller. The sending component (with the present ID) can act as a publisher and/or subscriber after the connection is established.
- $t(\text{ID})$ is sent by the component when terminating the connection.

- s(ID, topic) is used to order messages belonging to a specific topic. It is received by the controller and the sending component is put to subscribers for the topic.
- u(ID, topic) is sent by the given subscriber to unsubscribe from the given topic.
- p(ID, topic, retain, data) represents the "Publish" message sent by the given publisher in the first stage of the path, that is, from the membrane of the publishing component to the controller. The third parameter "retain" takes the value 0 or 1. The value 1 means that the data for the given topic should be stored, and forwarded to a component that sends order just after the publication of this data.
- d(publisherID, subscriberID, topic, retain, data) represents the same message, but in the second stage. The controller creates this type of object for each subscriber for the topic and adds the second parameter.

The objects c and t are intended for activation and deactivation of components. In the real world, this means establishing a session between a component and the controller (c, connect) and terminating the session (t). When establishing a session, the component authenticates itself (must pass credentials).

The object s is intended for ordering a subscription to a selected topic, and the object u serves to unsubscribe. Both objects need the sender's ID, and s contains the topic name to subscribe.

The objects p and d are used for the publish operation. We need two types of objects for this operation because we need to distinguish the two phases of the message path (from the publisher to the controller and from the controller to the subscriber). In the second phase, the subscriberID property is added to the object. The properties of p and d correspond to the publish message type: the sender's ID (publisherID), the topic to contribute. One of the properties is the "retain" value that determines whether the published data should be stored for newly connected subscribers, not just forwarded to current subscribers. And of course the data. For d we need one additional property, the target (subscriberID).

The use of the denoted objects is shown in Figure 3.

Let us define the evolution rules for each membrane. The rules are used to transport objects between membranes.

Assume that the controller has a database (denoted as *subscriptionsDB*) of all subscribers for each topic. For simplicity, we represent the mentioned database as a set of ordered pairs (topic, subscriber). Each topic can have multiple subscribers and each subscriber can subscribe to multiple topics.

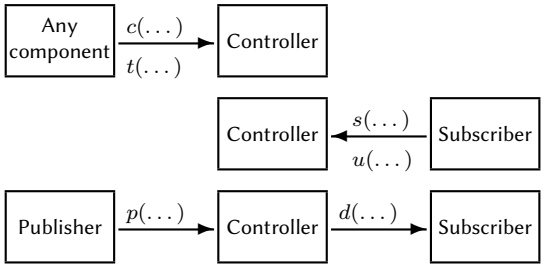

Figure 3: Objects for the operations connect/disconnect, subscribe/unsubscribe, publish

For the skin membrane (controller) we need the following set of rules: for each publisher i publishing in a topic t with the retain value r

$$p(i, t, r, \textit{data}) \rightarrow \bigcup_s d(i, s, t, r, \textit{data})_{here} \quad \forall (t, s) \in \textit{subscriptionsDB}$$

$$d(i, s, t, r, \textit{data}) \rightarrow d(i, s, t, r, \textit{data})_{in_s} \quad \forall \text{ subscribers } s$$

For each component i, topic t and retain value r:

$$p(i, t, r, \textit{data}) \rightarrow p(i, t, r, \textit{data})_{out}$$

These rules can also exist for the topics in which the component does not publish because the object on the left side of the rule appears in the membrane only when the component starts publishing data to the topic.

For each component i and each topic t

$$s(i, t) \rightarrow s(i, t)_{out}$$

We create these rules again for all possible topics.

Similarly, we need to transport other objects from the components i:

$c(i, \text{credentials}) \rightarrow c(i, \text{credentials})_{out}$
$t(i) \rightarrow t(i)_{out}$
$u(i, t) \rightarrow u(i, t)_{out}$ for all possible topics t

3.2. Control Layer

The role of the control layer is to mediate communication between the membrane system and devices.

If a device sends the message to establish a connection with the controller, the control layer creates an object with the device index and credentials in the corresponding membrane. If a device produces data and sends the "produce" message, the control layer creates an object with the appropriate parameters in the corresponding membrane. The procedure is similar when any other message occurs.

If an object indicating a message for the corresponding device appears in a membrane, the control layer reacts again: it removes the object from the membrane and passes the relevant message to the given device.

Algorithm 1: Messages and corresponding objects

```
// Parental message/object with one common
   property, all messages have a sender:
message:
   type,   // publish, deliver,...
   ID;     // ID of the source component

// Object generated by a publisher, going to the
   controller (the first phase of publishing message):
publish (child of: message): ..... p
   topic,
   retain,   // 0 or 1
   data;

// Object transformed by the controller, going to
   a subscriber (the second phase):
deliver (child of: publish): ..... d
   subscriberID;

// Subscribing message with an order:
subscribe (child of: message): ..... s
   topic;   // topic to subscribe

// Unsubscribing message for some topic:
unsubscribe (child of: message): ..... u
   topic;   // topic to unsubscribe

// Connection message for a component:
connect (child of: message): ..... c
   credentials;   // e.g. username, password

// Disconnection message for a component:
disconnect (child of: message) ..... t
```

Algorithm 2: Entities – properties

```
topic:
   topic,
   lastPublisher,
   lastValue;   // last published data

order:
   topic,
   subscriberID;

// Parental object for all entities:
entity:
   ID,   // identification number
   membrane,   // ref. to the corresp. membrane
   device;   // ref. to the corresp. device

component (child of: entity):
   turnedOn,   // 0 (false) or 1 (true)
   // Properties for publishing:
   publishTopic,
   publishRetain,   // 0 or 1
   // Property for subscribing:
   topic[] orderedTopics;

controller (child of: entity):
   component[] components,
   authenticator,
   topic[] topicsDB,   // registered topics
   order[] subscriptionsDB;   // subscriptions
```

The control layer does not deal with the forwarding of messages or the transfer of objects, the membrane system is in charge of these operations.

All the messages sent between components and their corresponding objects are shown in Algorithm 1. Each message/object has its sender, so all messages (objects) have the property ID (sender's ID) inherited from the parent message/object "*message*". Other properties depend on the type of message/object.

Algorithm 2 shows the properties of the controller and individual components. Each component can publish into one topic (*publishTopic*) and subscribe data to multiple topics (*orderedTopics*). All components have their corresponding membranes inherited into the skin membrane.

The controller registers topics that can be subscribed to (*topicsDB*), and all current subscriptions (*subscriptionsDB*). The corresponding membrane is the skin membrane.

In Algorithm 3 and 4 we can find the functions of the components and the controller. The components as the part of the control layer only perform the transformation between representation in the object form (for membranes) and the message form (for devices). For the publish message, if the protocol used by the given device does not provide working with topics, we can use the property *publishTopic*.

In contrast, the controller does not perform transformations, but works with databases whose list can be seen in Algorithm 2. If the controller device itself does not provide authentication, the control layer can do it. For new subscriptions, it is possible to use the retain property, and we can also add additional functionality if required.

The transformation between the objects p and d is made inside the membrane layer using the appropriate rule, not inside the control layer.

3.3. Membrane Firewall

A firewall is a traffic filter, i.e. it defines which communication is allowed and which is not. In our system, we can implement the firewall at any layer, it depends on where we want it to interfere with traffic. Putting the firewall in the membrane layer has the advantage that the firewall is harder to detect for a potential attacker and we have access to all traffic. However, we need the possibility to interfere with the evolution rules of membranes.

Algorithm 3: Components – functions

```
// An object received from the membrane:
function component.receiveObject(obj)
begin
    if obj.type = = d then
      device^.processMessage(deliver,
      obj.publisherID, obj.topic, obj.retain, obj.data) ;
end

// A message received from the device:
function component.receiveMessage(mes)
begin
    switch mes.type do
        case connect do
          membrane^.createObject(c, mes.ID,
          mes.credentials);
        case disconnect do
          membrane^.createObject(t, mes.ID);
        case subscribe do
          membrane^.createObject(s, mes.ID,
          mes.topic);
        case unsubscribe do
          membrane^.createObject(u, mes.ID,
          mes.topic);
        case publish do
          membrane^.createObject(p, mes.ID,
          mes.topic, mes.retain, mes.data);
    end
end
```

Algorithm 4: Controller – function

```
// An object received from the membrane:
function controller.receiveObject(obj)
begin
    switch obj.type do
        case c do
            if (authenticator.check(obj.ID,
              obj.credentials)) and
              (device^.processMessage(connect,
              obj.ID, obj.credentials)) then
                components[obj.ID].turnOn();
        case d do
            device^.processMessage(disconnect,
              obj.ID);
            components[obj.ID].turnOff();
        case s do
            device^.processMessage(obj.ID, obj.topic);
            subscriptionsDB.add(obj.topic, obj.ID);
            if topicsDB.retainSet(obj.topic) then
                membrane^.createObject(d,
                  topicsDB.lastPublisher(obj.topic),
                  obj.ID, obj.topic, 1,
                  topicDB.lastValue(obj.topic));
        case u do
            subscriptionsDB.remove(obj.topic, obj.ID);
            device^.processMessage(obj.ID, obj.topic);
        case p do
            if obj.retain then
              topicsDB.storeData(obj.topic, obj.data);
            else topicsDB.unsetRetain(obj.topic) ;
    end
end
```

In the previous sections, we assumed that all components can publish in any topic and can subscribe to any topic. Thus, a natural implementation of a firewall can simply be to restrict the set of evolution rules for certain topics and certain components according to the specified requirements. We can proceed in one of the following ways:

1. We intervene in the rule replacing object p with a set of objects d inside the skin membrane. The d object for a given subscriber and topic will or will not be generated.
2. We add or remove the rule transferring the object d into the membrane of the target component. In this case, it is advisable to create a deletion rule for the given object.

Whichever option we choose, the firewall does not delay traffic in any way. The published object is either transferred, ignored or deleted, always within one rule. Suppose we choose the first option.

The firewall must also be managed by adjusting evolution rules. We can provide this functionality in the control layer. The controller contains a database with settings for the firewall, according to which it reacts when receiving an object s (subscription):

```
if firewall.allow(obj.ID, obj.topic) then
    membrane^.adjustRules_addSubscription(obj);
```

The consequence is a modification of all the relevant rules transforming the object p with the given topic into sets of objects d (the object d for the given subscriber and topic is added).

Furthermore, for disconnect messages, the given object needs to be removed from the relevant rule. Therefore, if any rule transforming object p to d in the skin membrane is applied, only such d objects intended for any allowed communication are created.

4. Discussion

In this paper, we comply with the message types and overall functionality of the MQTT protocol, but only in a simplified way. The mentioned protocol actually

uses other types of messages (including confirmations), which of course can also be implemented as objects in membranes. Unfortunately, there is not enough space for the complete modeling of this communication.

The proposed firewall could be enriched with a quarantine membrane, and rules for the skin membrane that forward "suspicious" traffic to the quarantine membrane. A supervisor could regularly check this membrane and would gain an overview of risky traffic.

The goal of this paper is to use a mechanism independent of, although inspired by, specific existing protocols to propose communication between IoT devices. A membrane system was used as a basis, which can be further elaborated: there are programming languages for implementing membrane systems, and other languages for network applications can be used as well.

Acknowledgments

This work was supported by the project no. CZ.02.2.69/0.0/0.0/18_054/0014696, "Development of R&D capacities of the Silesian University in Opava", co-funded by the European Union.

References

[1] G. Păun, Membrane Computing: An Introduction, Springer, Heidelberg, 2002.

[2] G. Păun, G. Rozenberg, A guide to membrane computing, Theor. Comp. Science 287 (2002) 73–100.

[3] G. Păun, G. Rozenberg, A. Salomaa, The Oxford Handbook of Membrane Computing, Oxford University Press, New York, 2010.

[4] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Computing 3 (2016) 76–83.

[5] V. Sharma, K. Srinivasan, D. N. K. Jayakody, O. Rana, R. Kumar, Managing service-heterogeneity using osmotic computing, in: International Conference on Communication, Management and Information Technology (ICCMIT 2017), Warsaw, Poland, 2017.

[6] A. Buzachis, D. Boruta, M. Villari, J. Spillner, Modeling and emulation of an osmotic computing ecosystem using osmotictoolkit, in: 2021 Australasian Computer Science Week Multiconference, ACSW '21, Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3437378.3444366. doi:10.1145/3437378.3444366.

[7] S. K. Datta, C. Bonnet, Next-generation, data centric and end-to-end iot architecture based on microservices, in: IEEE International Conference on Consumer Electronics – Asia (ICCE-Asia), 2018, pp. 206–212. doi:10.1109/ICCE-ASIA.2018.8552135.

[8] S. Vavreckova, Modeling communication in internet of things network using membranes, in: CEUR Proceedings of the 21st Conference Information Technologies - Applications and Theory (ITAT 2021), 2021, pp. 195–201.

[9] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[10] N. Busi, Causality in membrane systems, Membrane Computing (2007) 160–171.

[11] W. Kassab, K. A. Darabkh, A–z survey of internet of things: Architectures, protocols, applications, recent advances, future directions and recommendations, Journal of Network and Computer Applications 163 (2020). URL: https://doi.org/10.1016/j.jnca.2020.102663.

[12] J. Dizdarević, F. Carpio, A. Jukan, X. Masip-Bruin, A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration, Association for Computing Machinery 51 (2019). URL: https://doi.org/10.1145/3292674. doi:10.1145/3292674.

[13] P. R. Egli, MQTT – Message Queueing Telemetry Transport, Zurich University of Applied Sciences, Zurich, 2017. doi:10.13140/RG.2.2.13210.54721.