Smart Home

[Introduction 2](#_Toc67232664)

[Database 2](#_Toc67232665)

[Web Server 4](#_Toc67232666)

[Environment 4](#_Toc67232667)

[Project Management 4](#_Toc67232668)

[Source Code 4](#_Toc67232669)

[Design 5](#_Toc67232670)

[Modules : 5](#_Toc67232671)

[System Manager 5](#_Toc67232672)

[Communication Manager 5](#_Toc67232673)

[Database Manager 12](#_Toc67232674)

[App manager 13](#_Toc67232675)

[Users Manager 13](#_Toc67232676)

[Libraries 13](#_Toc67232677)

[Security 13](#_Toc67232678)

[Quick Startup Guide 13](#_Toc67232679)

[Resources 13](#_Toc67232680)

## Introduction

The objective of this project is to build a generic infrastructure to host a smart home applications.

## Database

Postgresql will be the database used in the projects to store all the program data. For C/C++ applications Postgresql C/C++ Interface API will be used.

sudo apt-get install postgresql

*performing post-bootstrap initialization ... ok*

*syncing data to disk ... ok*

*Success. You can now start the database server using:*

*/usr/lib/postgresql/10/bin/pg\_ctl -D /var/lib/postgresql/10/main -l logfile start*

Use systemctl command to manage postgresql service:

1. stop service:

systemctl stop postgresql

1. start service:

systemctl start postgresql

1. show status of service:

systemctl status postgresql

1. disable service(not auto-start any more)

systemctl disable postgresql

1. enable service postgresql(auto-start)

systemctl enable postgresql

/var/lib/postgresql/10/main

Postgresql version = 10

config files : /etc/postgresql/10/main/

log file : /var/log/postgresql/postgresql-10-main.log

<http://manpages.ubuntu.com/manpages/eoan/en/man1/pg_lsclusters.1.html>

Latest Postgresql C++ client code : <https://github.com/jtv/libpqxx>

Postgresql tutorial : <https://www.tutorialspoint.com/postgresql/postgresql_c_cpp.htm>

psql -U postgres

libtool: finish: PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/sbin" ldconfig -n /usr/local/lib

----------------------------------------------------------------------

Libraries have been installed in:

/usr/local/lib

If you ever happen to want to link against installed libraries

in a given directory, LIBDIR, you must either use libtool, and

specify the full pathname of the library, or use the '-LLIBDIR'

flag during linking and do at least one of the following:

- add LIBDIR to the 'LD\_LIBRARY\_PATH' environment variable

during execution

- add LIBDIR to the 'LD\_RUN\_PATH' environment variable

during linking

- use the '-Wl,-rpath -Wl,LIBDIR' linker flag

- have your system administrator add LIBDIR to '/etc/ld.so.conf'

See any operating system documentation about shared libraries for

more information, such as the ld(1) and ld.so(8) manual pages.

----------------------------------------------------------------------

make[2]: Nothing to be done for 'install-data-am'.

## Web Server

Apache Tomcat

## Environment

The entire project will be hosted inside a docker container.

The project image “smarthome” can be downloaded from the below docker hub.

<https://cloud.docker.com/repository/docker/pruthwinkg/dockerhub-pruthwinkg/general>

## Project Management

The project management tool used for this project is “Trello”.

Below is the link of the project management activities for this project.

<https://trello.com/b/OG8rQ0q8/smart-home-system>

## Source Code

The source code of this project is maintained in the “Bitbucket” repository.

<https://bitbucket.org/Pruthwinkg/hobbyprojects/src/master/SmartHome/>

apt-get install libpq-dev

apt-get install libboost-all-dev

## Design

### Modules:

#### System Manager

System Manager allocates UIDs for each sub-system. The UIDs can be allocated statically or dynamically. There are 1024 well known static UIDs defined. All the sub-systems required for the smooth functionality and support of the system is allocated a well known UID. Most of the user apps are allocated dynamic UIDs.

The Application Manager which is responsible for handling user apps borrows the dynamic UIDs from the System Manager.

These UIDs are required for Communication Manager during the protocol phase.

Future Request for this module:

- The deletion of clients should be more smarter. There has to be a option to notify the clients, so that the clients can finish and respond back to the sysmgr that they are ready to be deleted. This notification can be sent over SCOM messages with a super priority.

#### Communication Manager

<https://opensource.com/article/19/4/interprocess-communication-linux-storage>

<https://opensource.com/article/19/4/interprocess-communication-linux-channels>

<https://opensource.com/article/19/4/interprocess-communication-linux-networking>

This subsystem will handle all kinds of communications (within the system and to outside).

From the articles above, the Unix Domain Sockets (UDS) suits well for this application. The Comm manager will also implement other IPC mechanisms as well incrementally. But the primary mode of communication will be via Unix Domain Sockets.

The comm\_mananger will implement below features :

- Message Queuing

- Priority based message delivery

- Direct channel establishment for bulk data delivery between two modules (Like DMA)

- Backing of messages (in case of restarts/crashes)

- Stream and Datagram support

- Event Handling (Does this has to be handled by a new subsystem, like Event Manager ???)

The UDS Server will always run in the context of comm\_manager. The comm\_mananger will provide a shared library to create clients for each subsystems who wants to participate in the communication. The UDS clients will run in the context of each subsystem.

**Note : Eth and IP header is valid only for communication outside the system**

|  |  |  |  |
| --- | --- | --- | --- |
| Eth | IP | Communication Manager Header | Pay load |

**Communication Manager Protocol:**

Communication Manager is an intelligent module whose main responsibility is to transfer data from process to another. This module supports numerous features. The Communication Manager has two parts i.e, Server Instances and Library part.

Communication Manager uses a custom protocol for reliable data transfer. It has mainly three stages. The protocol is always initiated by the communication manager. It not necessary that all these steps happen sequentially. The Server Instance and the Communication Manager library follows a series of steps before allowing any regular data transmission. The library tries to offload many data storing, so that Communication Manager remains faster. But Communication Manager can still store some data packets if required.

**Discovery Phase**:

Every process will be identified by an UID. (Statically allocated/ Dynamically allocated by System Manager, written to mmap). Discovery phase is usually kick started when an app uses the communication system for the first time. If an app never communicates with anyone, none of the protocol kicks in. During this phase, the Communication Manger discovers that such a process exists in the system and makes a note of it.

The Communication Manager library maintains a state machine to know the Communication state of an app. When an app wants to send messages/data to another app, the Communication Library allows the data to be sent only if the current state of the app is in Data Transfer ready. The library will hold the data in its internal queues and kick starts the protocol by sending a copy of this data packet to the Server Instance.

Upon receiving the data packet from the app for the very first time, the Server Instance will initiate the Discovery protocol. During the discovery phase, the Communication Manager will send a discovery msg to the client, from where it received the message for the very first time. The Communication Manager also sends some system wide information in the Discovery messages for the app to consume. It’s up to the app, to make use of this system wide data.

The client needs to send an ACK for the Discovery message. The client also has to respond back to the Communication Manager with a Discovery message along with its UID.

Now the Communication Manager maintains a map of <UID,Server FD>. This indirectly helps in the Learning phase as well for other apps who wants to send messages to this app.

Now the Communication Manager library of the app will set the state of the app to “Data Transfer Ready” and will send the data packet which was enqueued in its internal queues to the Server Instance. If the destination UID is also discovered by the Server Instance, then the packets are sent. Else, the Server Instance will notify the library to hold off any data packet sending to this particular destination UID, until it is Discovered (Learning phase).

“Data Transfer Ready” doesn’t necessarily mean that the Data packets can reach the destination apps. It just means that the source app is ready for Communication. But it guarantees that it can receive the packets from other apps. Till the Destination app is also Discovered by the Communication Manager, the source app cannot send any data packets to that destination.

The library will make a note that, this particular destination UID is not yet reachable and will start queuing the data packets if the app keeps sending the data packets to this destination. The library can also ask the app to throttle the messages in case lot of messages for this destination starts to build up in its internal queues. The queues are per destination UID.

**Learning:**

The learning phase is indirectly assisted by the Discovery phase. As mentioned in the previous phase, if a destination UID is not yet reachable, the library will stop sending any data packets for this destination to the Communication Manager. Soon after the Discovery phase, when the library sends a data packet again to the Communication Manager, if the destination UID is not learnt/Discovered, the Communication Manager will send a learning message with payload as NULL followed by the destination UID. Once the Destination app is also discovered by the Communication Manager, it will send Learning messages to all those apps who wanted to send data to this destination previously. After the Discovery phase, when an app sends data packets for a destination who is not yet discovered, the Communication Manager will maintain an interest table to know who are interested to send data to this destination.

The learning message is a simple message without requiring any ACK. Upon receiving this learning message, the Communication Manager library will start sending messages which were queued up in its internal queues for this destination. The payload of the learning message will contain the destination UID if that destination has been discovered/learnt by the Communication Manager.

Since the learning message doesn’t require ACK, after sending the learning messages to the interested apps, the Communication manager will clear the interest tables for this destination UID. If an app misses the learning message, then there is no way to notify the app unless the app tries to send the message to the Communication Manager again.

The library can alternatively send a special system message to the Communication Manager asking whether the Destination is learnt/discovered.

**Data Transfer:**

After the source UID and destination UID are both discovered/learnt by the Communication Manager, an app can actually send data to that destination UID. If Communication Manager sends learning message with payload as NULL followed by a destination UID, at any point of time, the clients should hold all its further messages to that destination UID. Else, the Communication Manager drops those messages.

The Communication Manager can simply decide to throttle the messages from a source app, in case if the Communication Manager is extremely loaded OR it receives extreme large quantities of messages from this source app OR if a destination is not yet discovered/learnt. To throttle the messages for a particular destination, it will simply send a learning message with NULL followed by a destination UID in its payload. The client app should stop sending any further messages for any destination till receives again a learning message with destination UID in its payload. The Communication Manager can also send a special system message to indicate the app to stop sending to any destination.

|  |  |
| --- | --- |
| **Message Format** | **Functionality** |
| Protocol | Discovery | <system info> | During discovery phase |
| Protocol | Learning | <NULL, Destination UID> | Destination not learnt. Stop sending to this UID |
| Protocol | Learning | <Destination UID> | Destination learnt. Start sending to this UID |
| ACK | Proto ack/nack | | Acknowledgment for Protocol packets |
| ACK | Data ack/nack | | Acknowledgment for Data packets |
| System | Communication | <msg id> | Refer system messages section for more details about this type of messages.  Depending on msg\_id value, the messages can be throttled for all destinations from this source UID |

The Communication Manager and its library will support APIs to encode and decode messages required for the protocol.

**Multi-Master instances:**

The Communication Manager supports multi-master instances. These are the instances of the Communication Manager, but all of them runs in the context of Communication Manager. There is a software limit of max 5 master instances in a system, due to memory and computing limitations of the underlying hardware on which the system runs. Each Master instances can belong to different AF types or can also create multiple masters of the same AF types. All these Master instances needs to be created at the compile time only. Some of the types of master instances are UDS (Unix Domain Sockets), V4/V6 IDS (Internet Domain Sockets). UDS types of Masters are used for intra-communication and IDS can be used for both inter and intra system communication. Depending on the Communication types needed by the client, connection need to be setup between the client and the Master instances using the Communication Manager Protocols.

Same type of Master instances are used to share the load, by mapping the UIDs to the Master instances of the same type. For example, if two UDS Masters are created, one Master can be responsible for serving static UIDs and other Master instances for all Dynamic UIDs. Even a set of specific static UIDs can be mapped to a Master instance.

All the Master instances regardless of the type share many internal tables like UID table. This allows to avoid running the protocol for each master instances again for a UID which allows for faster convergence. If a UID has already been discovered/learnt by a Master Instance, that knowledge can be used by other Master instances to start serving the UID as fast as possible. Also, every master instance run on its own set of worker threads, has its**z** own send/recv/protocol/housekeeping queues.

To reduce the number of workers threads, an option called Load sharing is added to the Communication Manager to share many worker threads in multi-master scenario. With this option enabled, at least the request handler should be run on separate worker thread for every master instance. But the other set of worker threads will be shared by master instances. This option is available only for master instances of the same type.

For applications using the Communication Manager library, it can create multiple clients for example one for sending/receiving regular data and other client for ancillary communication depending on the use case of the application. In case if an application is required to create multiple clients, then it needs to create at least two dedicated threads for the communication handler purpose. Other tasks like housekeeping/application data receivers can be made to share the worker threads (just like the load sharing feature on the Server side) by querying for both the client IDs. (Refer Communication Manager Library Test app code)

**UDS:**

Unix Domain Socket Master instance is used for intra system communication. Two Master instances are created for UDS, Default and Secondary. For most of the communication Default instance is used. Secondary instance is used for Communication ancillary data like passing file descriptors, passing credentials etc. Secondary UDS instance is even capable of packing and sending multiple user data in a single send. But for simplicity purpose, default instance is recommended for regular data communication. Load sharing is enabled to reduce the number of worker threads in UDS.

**IDS:**

IDS is mainly used for inter-system communication. The IDS has a sender/receiver role. When a client wants to send a message to a client in another system, it sends the message first to the IDS on the same system. The IDS now sends this message to the IDS receiver on the other system, which in turn will deliver the message to the dest client. There is a different protocol between the IDS of the different systems for reliable connection

**Communication Manager Ancillary message support:**

Communication Manager also supports sending ancillary messages like file descriptors, credentials with appropriate permissions. This type of message also supports sending multiple user payload in one send/recv call. Only User data is allowed via the Ancillary Message as of now.

The message format is as below (logical representation):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Eth | IP | Communication Manager Header  (Msg Type = Ancillary) | Ancillary message header | Pay load |

The Ancillary messages includes Communication Manager Header which has the target information like src\_uid, dst\_uid and other fields which are also applicable here. The msg\_type is always set to Ancillary. The Ancillary Msg header has message type which tells about the payload it is carrying.

The ancillary messages are very bulky in nature. So, it is recommended to be used to send either file descriptors OR to do a bulk data transfer.

**Note**: Ancillary Communication cannot exist as standalone in a client application. They need to co-exist with the Normal mode of Communication. Ancillary messages don’t participate in Communication Manager protocol. Hence the Ancillary Communication needs the help of Normal Communication channel to bypass the protocol and make their end points discoverable by the Communication Manager. The Communication Manager uses Protocol Message with a sub type as Ancillary Learning to learn about the Ancillary Server FDs corresponding to the UIDs. When client A wants to send an Ancillary message to client B, both the clients should have been discovered by the Server. Now when the Server receives the Ancillary message from A destined to B, it first populates the existing protocol table of A with the Server FD corresponding to the Ancillary Communication channel. If the protocol table doesn’t exist for A or B, the Server will simply drop the Ancillary message. Now the Server will send a Protocol Message with a sub type as Ancillary Learning to Client B asking it to send a dummy Ancillary message with the Dest UID of Communication Manager itself. When the Client B responds with an Ancillary message, the Server will populate the protocol table entry of B with the Server FD. Now the Server will forward the saved Ancillary message to B, using the Server FD it has learnt. From this point onwards, the Ancillary message Communication will be faster since the Server has learnt the Server FDs of Ancillary Communication channel of both the clients. If the Client B, doesn’t have Ancillary Communication open, it will send back a negative response. In that case, the Server will drop the saved Ancillary message from Client A and inform it that client B doesn’t have the capability to receive Ancillary message.

**Communication Manager Overlay Protocols:**

Communication Manager allows to implement custom overlay protocol messaging on top of it. One such fine example is SCOM.

**System Messages:**

These are special messages sent only by the infra modules to other infra modules or user apps. These are usually high priority messages on which the recipients have to usually take actions. One example of system messages is, which sent by the Communication Manager to throttle the sending frequency.

Format of system messages:

[Msg\_type:<System message>] [submsg\_type: <Module/Functionality>] [Payload:<Msg-Id>, <Optional Message>]

The message IDs will be specific for a Module/Functionality.

**SCOM**

“SCOM” (Smart Communication) uses Communication Managers Custom Overlay Protocol. SCOM will provide library APIs to communicate with other subsystems.

The SCOM messages will consists of a header and a payload. Sometimes the payload can be optional. The header will include some of the fields like source address, destination address, payload length, message type etc

The source and destination addresses are allocated for each and every process in the system either statically or dynamically. The addresses of all the non-user modules are allocated by the system manager statically. The addresses of all the user apps are allocated by the app manager dynamically (So to add a new user app, the app manager need not to be restarted. The new user app will send SCOM messages to app\_manager trying to register itself with the system. The app\_manager then will allocate it an address and broadcast all the user apps with an updated list of addresses of user apps)

**Event Management**

<https://stackoverflow.com/questions/2533284/are-posix-pipes-lightweight>

<http://man7.org/linux/man-pages/man2/eventfd.2.html>

<https://stackoverflow.com/questions/13587003/select-doesnt-behave-properly-with-an-eventfd>

#### Database Manager

#### App manager

#### Users Manager

### Libraries

There are few libraries available which provides many rich functionalities to the infra modules as well to the user applications.

#### Interface Library

Interface library is used to provide a standard way of interfacing with the modules or apps. This makes the implementation in a uniform way. The interfaces are like a gateway via which the modules can talk to any other modules who also use this library. The modules don’t have to care, how the actual communication happens as it writes/reads from the interfaces.

The modules can create as many as interfaces it wants. The modules can block, allow any interfaces it owns at any point of time. The interfaces also have various security levels which the apps can assign/modify any point of time. The interfaces also have small buffers associated with it which helps the apps to read/write at their own pace till these buffers are full.

* Used to expose/config data to UI apps like shell
* Communication between various infra modules
* Communication between infra and user apps (Non-UI)
* Communication between user apps

### Security

## Quick Startup Guide

- Download the image “smarthome” from the dockerhub to a Linux machine.

- Spwan a container and expose the port 8080 to one of the ports on the Host machine.

*docker run -dit -p 8888:8080 --rm --name=SmartHome pruthwinkg/dockerhub-pruthwinkg:smarthome*

- Set up the “Port Forwarding” rule on the home router to allow the external requests to the Smart Home system.

## Resources

<https://www.tutorialspoint.com/postgresql/postgresql_c_cpp.htm>

<https://www.tutorialspoint.com/postgresql/postgresql_java.htm>

<https://docs.docker.com/engine/reference/commandline/docker/>

<https://gist.github.com/Alexey-N-Chernyshov/4634731>

<http://poincare.matf.bg.ac.rs/~ivana/courses/tos/sistemi_knjige/pomocno/apue/APUE/0201433079/ch17lev1sec4.html>