Smart Home

Table of Contents

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`
2.start service:
`systemctl start postgresql`
3.show status of service:
`systemctl status postgresql`
4.disable service(not auto-start any more)
`systemctl disable postgresql`
5.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	Comm Hdr	Pay load

Communication Manager Protocol:

Communication Manager uses a protocol to send and receive from one process and to transfer it to another. It has main three steps. The protocol is always initiated by the communication manager.

Discovery:

Every process will be identified by an UID. (Statically allocated/ Dynamically allocated by System Manager, written to mmap). During the discovery phase, after the UDS connection is established, the Communication Manager will send a discovery msg to the client, from where it received the message

for the very first time. The client will respond back to the Communication Manager with the UID. Now the Communication Manager maintains a map of <UID,Server FD>. Now the Communication Manager will send “Ready to Communicate” message to that client.

Learning:

After the Discovery phase, now the learning phase starts. At this point the client can start sending regular data to other sub-systems. If client A wants to send the data to client B, then in the message header it will embed the source and destination UIDs and send the message to the Communication Manager. The Communication Manager, upon receiving this message, looks in the map <UID, Server FD> to see if B is also present. In such a case, it will send the message to B using the server FD of B. In case client B, is not yet discovered, it backs up the message for future send and also sends a message back to A, saying “Message will be sent Later”. The message backing can have variable timeouts which is decided by A in the header. Once the message is delivered to B and B acknowledges, the Communication Manager will send the ACK to A. Even the ACK can be optional depending on A’s header.

Data Transfer :

After the Learning phase, now the client can send regular data to other sub-systems. Till the Communication Manager sends the client its “Ready for Data Transfer”, the clients should hold all its further messages. Else, the Communication Manager drops those messages

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.

All the Master instances of the same type share all the global queues like protocol queue etc. 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.

Load Sharing using same type Master-instance:

The same type of Master instances are created to share the load of communication. Lets take the example of UDS Master type. A system can have “N” number of same type of Masters. There is always a default UDS/IDS Master. All the clients connect to the default at first. The Master will respond the client saying who will start serving it. In most of the cases, when the number of UIDs and load is less, the Default Master always servers the client. In case, Communication Manager feels that the Default is going to be overloaded in future, for the next new client, the Default Master will ask the client to talk to a different UDS Master. The default UDS Master will also send the ID of that Master. The client now needs to close the communication with the Default immediately and create a connection with the new Master. The Communication Manager library knows how to interpret the ID that the default Master has sent. So that for the client, create/send/receive library calls are exactly same for all Masters.

<Future feature>

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 Overlay Protocols:

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

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

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/