CS321: Friday Tutorial

Tushar Semwal

MQ Telemetry Transport (MQTT)

MQTT: Basics

- A lightweight messaging protocol
- Publish/Subscribe
- For M2M telemetry with low-bandwidth and -footprint
- Created by IBM in 1999 for Oil pipeline telemetry via satellite. NOW OPEN SOURCE

By the way

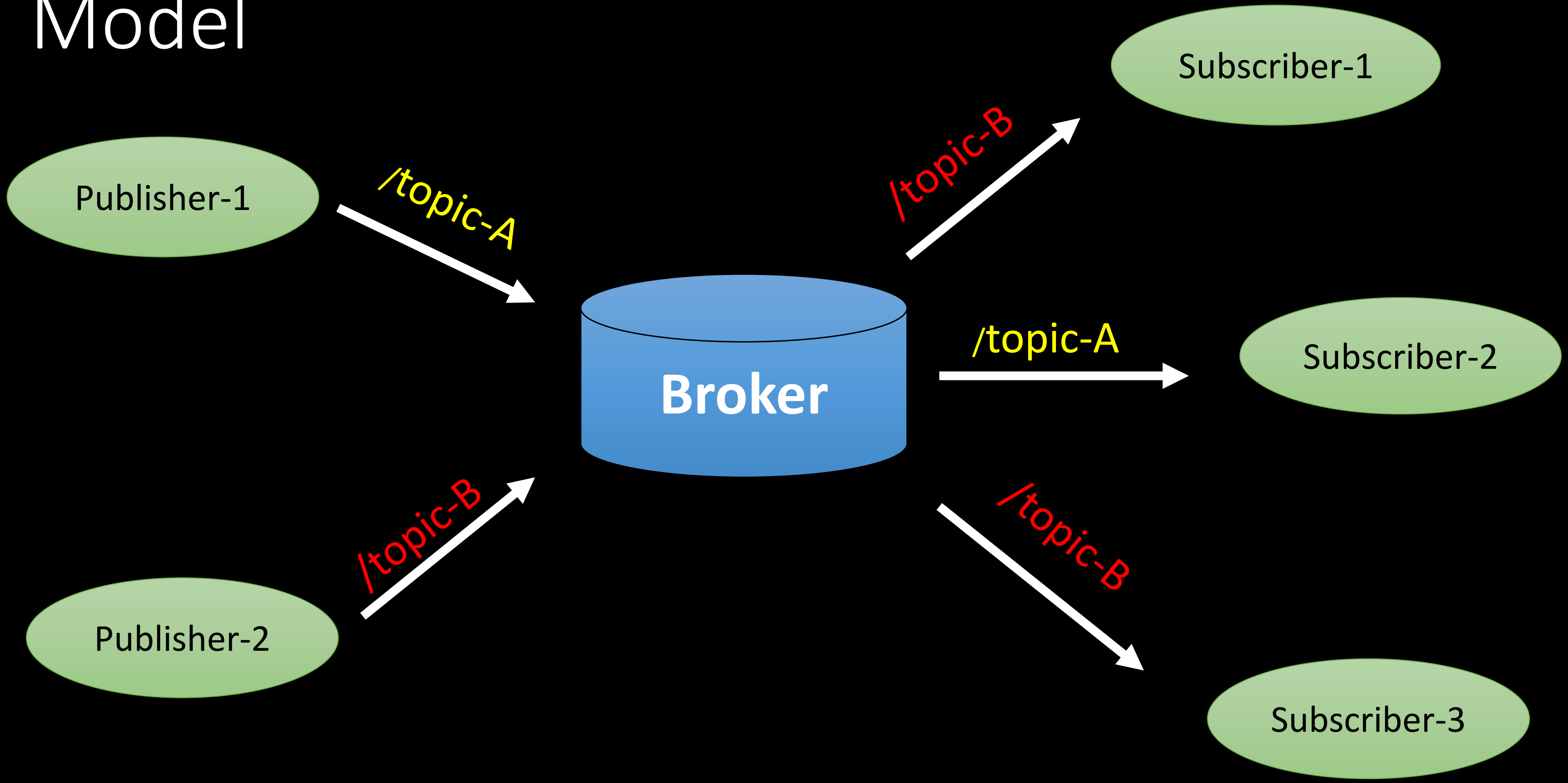
- Telemetering means??
- Tele + Metering
- Tele = Far or Remote
- Metering = Measurement

- Remote Measurements and sending data to the base server

Terms

- Client: A “device” that publishes a message or subscribes to a topic
- Publish: A client sends a message
- Subscribe: The broker sends the message about the topic to which a client is subscribed to.
- Topic: A namespace (casually, address string) to/from which clients publish and subscribe
- Broker: A “server” which accepts messages and delivers messages from/to clients

Model


Publisher-1
/topic-A
Publisher-2
/topic-B
Broker
/topic-B
Subscriber-1
/topic-A
Subscriber-2
/topic-B
Subscriber-3

Why not HTTP?

- Lot of header. Too big. MQTT smallest packet size is just 2 bytes
- Request/Response
- Usually synchronous. MQTT is asynchronous.

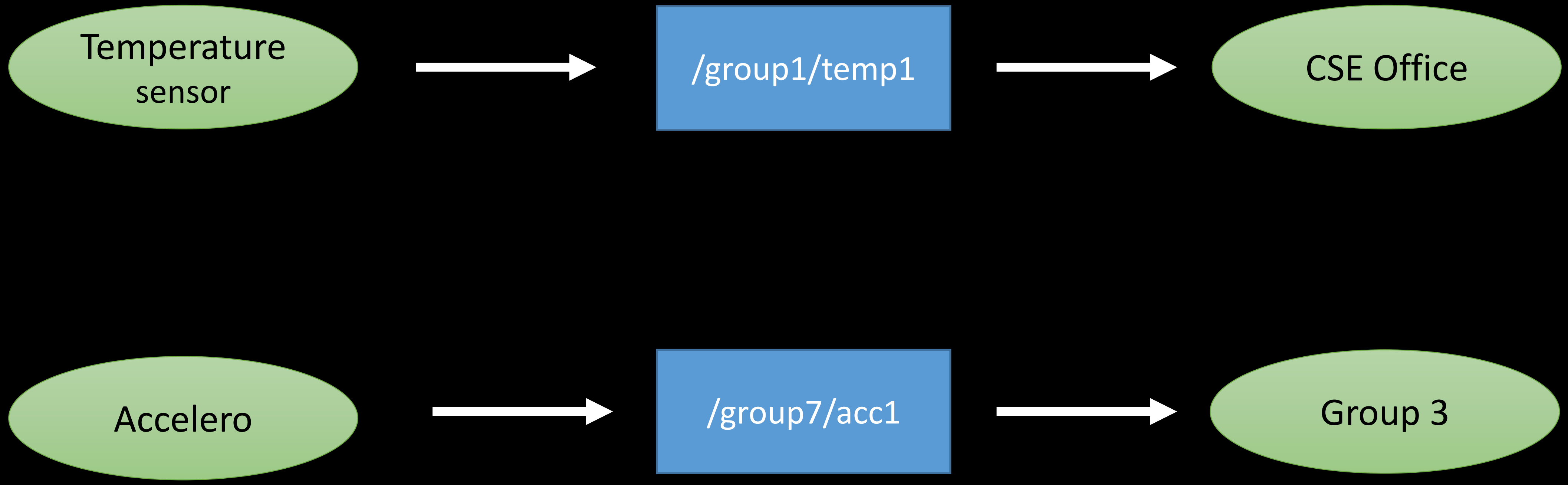
Usage

- Facebook Messenger (mobile app) uses MQTT to minimize latency and battery usage (https://ibm.co/2vNa8Uk)
- St Jude Medical, who use MQTT to remotely monitor patient implants (https://ibm.co/2JfCNWK)
- Consert, use MQTT as a part of their real-time home energy monitoring and management solution

Broker

- Open source implementations are available:
- Mosquitto (https://mosquitto.org/)
- Mirco Broker (https://github.com/micro/go-micro)

CS321: Use Case



Resouces

- https://www.baldengineer.com/mqtt-tutorial.html
- https://github.com/256dpi/arduino-mqtt

Embedded Programming

What is so different?

Difference

PC/Mobile

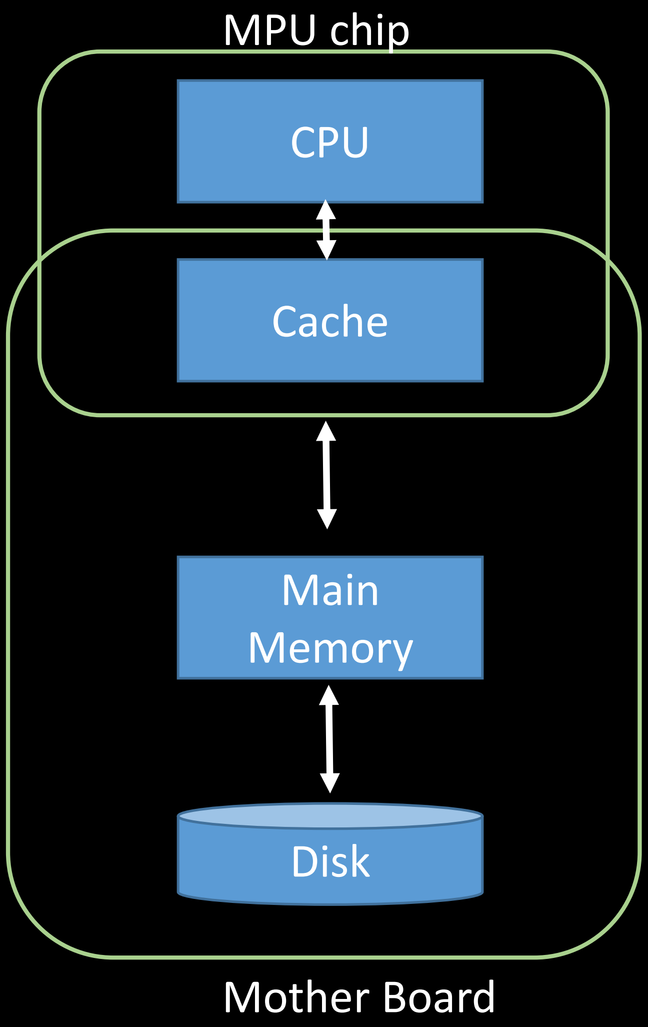
- General purpose
- Over 1MB of memory
- High performance CPU
- No energy constraints
- Goal: overall performance

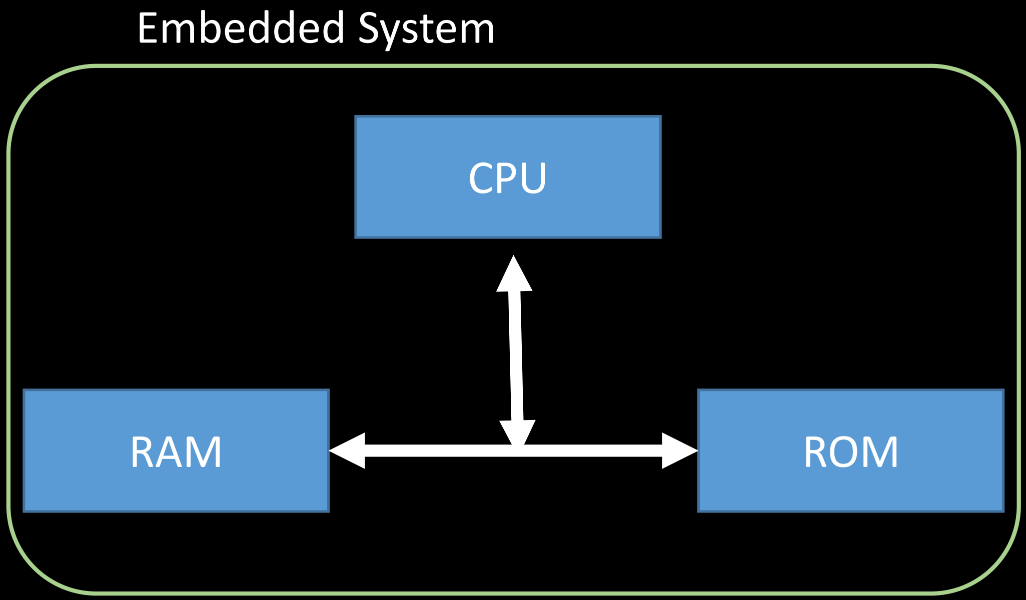
Embedded System

- Specific digital control
- Memory in Kbs
- Low-end MCU
- Energy constraint
- Goal: Attain required performance at the lowest cost

Why Memory is limited?

- Few variables are enough for digital control
- Less area
- Low cost
- Less energy
- Address <= 16 bit



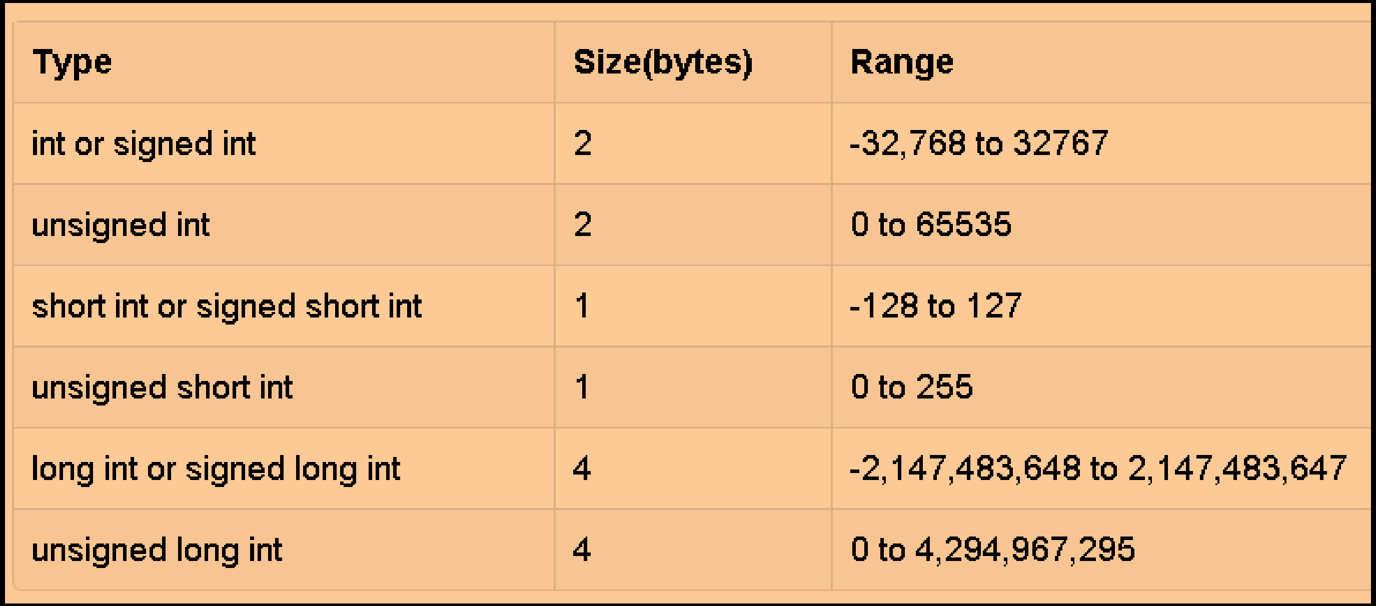


Power Consumption

```
int main(void)
{
Initialize();
while(1)
  {
          do_some_useful_work();
          hibernate(2ms);
  }
}
```

Choosing the Right Data Type

- Memory: Smaller variables
- Storage: Smaller constants and program



Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

```
<stdint.h>

uint8_t
uint16_t
uint32_t

int8_t
int16_t
int32_t
```

Floating point

- Be very careful
- Usually NO FPU
- Software managed
- Avoid usage if possible

Check memory usage with Arduino

- handson

Takeaways

- Use portable fixed size types
- Use smallest type possible
- Use floats only if necessary

Defining Qualifiers is IMPORTANT!!

- Qualifiers determine where the variable is stored and other stuffs:
  - Memory: Stored in RAM
  - Storage: Stored in ROM
  - Hardware: letting hardware directly access some variables for e.g. registers
  - Compiler optimization control

Quiz

```
int a;
int b;
void experiment(){
a=8; b=a*7;
If(a==8)
        printf(“a equals 8”)
else
        printf(“a NOT equals 8”)}
```

- What is the ouput?
  a) a equals 8
  b) a NOT equals 8
  c) Can’t say – I am a loser =D
- Is it possible to execute the ELSE part?
  a) YES
  b) NO

Quiz – why?

```
int a;
int b;
void experiment(){
a=8; b=a*7;
~~If(a==8)~~
        printf(“a equals 8”)
~~else~~
~~        printf(“a NOT equals 8”)~~}
```

- Multiprocessors with shared memory
- Multithreading
- Hardware attached variables Or memory mapped I/O
- Interrupts

Volatile Qualifier – Quick fix

```
volatile int a;
int b;
void experiment(){
a=8; b=a*7;
If(a==8)
        printf(“a equals 8”)
else
        printf(“a NOT equals 8”)}
```

Takeaways

- The volatile qualifier informs the compiler that variable may change because of hardware or other means
- Use to explicitly avoid optimization

Constants – const qualifier

- const int ledPin = 13;
- #define ledPin 13

- const stored in ROM
- #define also stored in ROM but copied wherever used in program

- #define simply replaces text
  - GOOD: easy to change the value of a constant
  - BAD: code bloating and no type or syntax checking – runtime errors!!
- const send to ROM once – memory addressing
  - GOOD: Large constants will be stored once, better for double, long
  - BAD: practically NONE

Function Alternatives

- Memory: Traditional functions are put in stack

- Storage: Look Up Tables (LUT) and Inline functions are stored in ROM
-
- Processing power: LUT are easy on CPU

Look Up Tables

- Constant Arrays containing a collection of return values
- e.g. const float log_LUT[256] = {-1.0E-30, 0.0000, 0.693147, ….}

- Multiplication tables we learned
- Some scientific calculators have LUTs

- uint8_t x;
- float y;

- y = log(x);
- y = log_LUT[x]
- e.g. IMU Euler angle calculations

Macro Functions

- #define square_macro(x) x*x
- Advantages
  - No need to send or return values – a bit fast
  - Readability
- Disadvantages
  - Code bloating

Inline Functions

- While regular functions may take time to execute AND Macros can be troublesome
- Inline provides best of both
- Identical to regular functions
- Just write inline while writing function definition

- Advantages over Macro
  - Parameters are inspected
  - Debugging is easier

Forced vs Suggested Inlining

```
int square_AI(unsigned char x)__attribute__((always_inline));
//Always Inline
int square_AI(unsigned char x){
return x*x;}

//suggested Inline
int inline square_SI(unsigned char x){
return x*x;}
```

Takeaways

- Macros are okay for simple functions
- If you need inlining, you may force it on compiler
- Otherwise leave inlining to compiler

Thank You!

Questions?