Flight Software Design

Michael Starch
NASA Jet Propulsion Laboratory
2022-10-17

Jet Propulsion Laboratory
California Institute of Technology

Overview of this Lesson

- Software Systems Design
- Systems Breakdown
  - Functionality
  - Interfaces
  - Data and Data Path
  - Off-Nominal Conditions
- Design Considerations
  - Initialization and Allocation
  - Deadlines, Timeliness
  - Concurrency, Threads
  - Faults, FATALs, and Error Handling
- Modeling Flight Software in F´
  - Topologies, Components, and Ports
  - Data Serialization
- F´ Design Patterns
  - Adapter
  - Manager - Worker
  - Rate-groups Timeliness
  - Hub Pattern

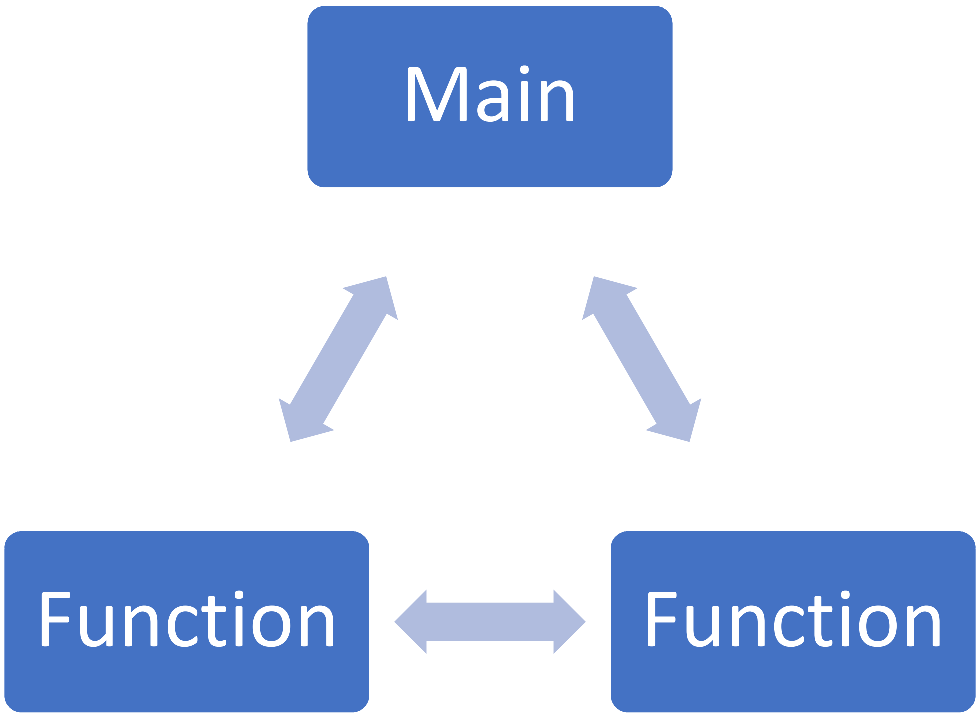
Software Systems Design

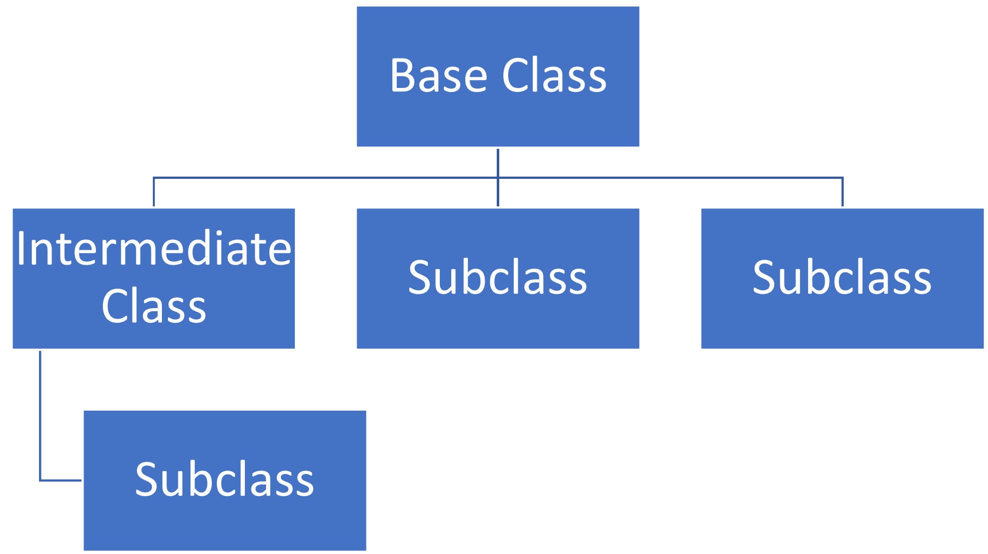
Software as a System

- Software is composed of more than one units ***components***
- Components are composed into a system or ***topology***
- Essential to understand ***topology*** before designing components

- Examples:
  - Python uses composition of modules
  - Java is organized through class compositions
  - F´ uses component topologies

Software as a System (Static)



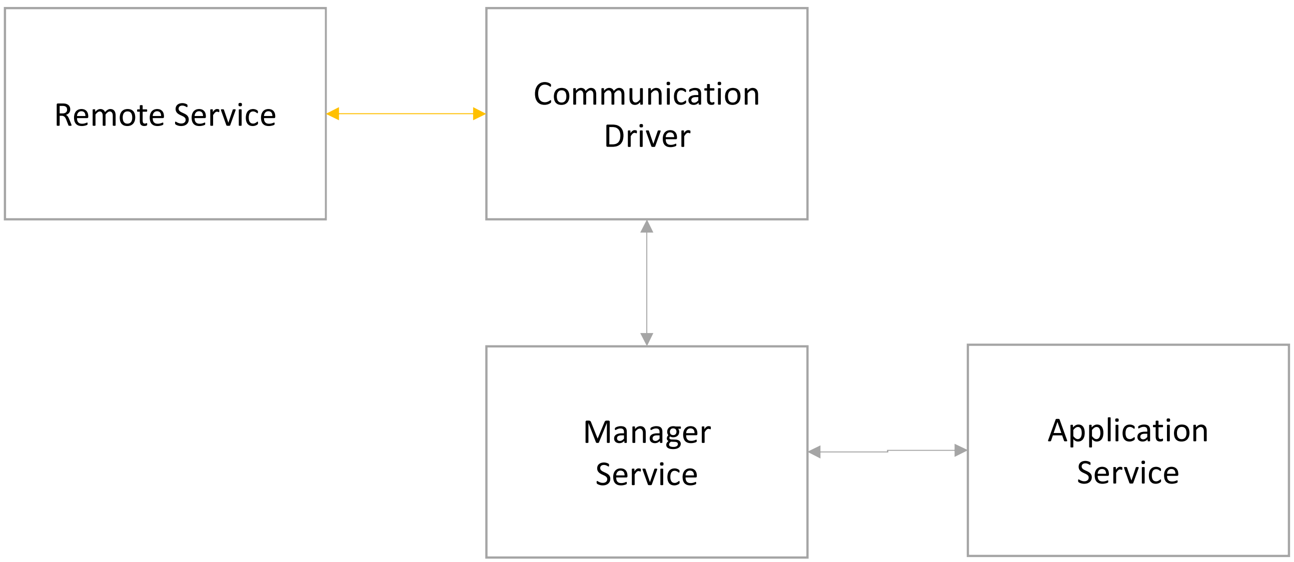


Software as a System (Static)

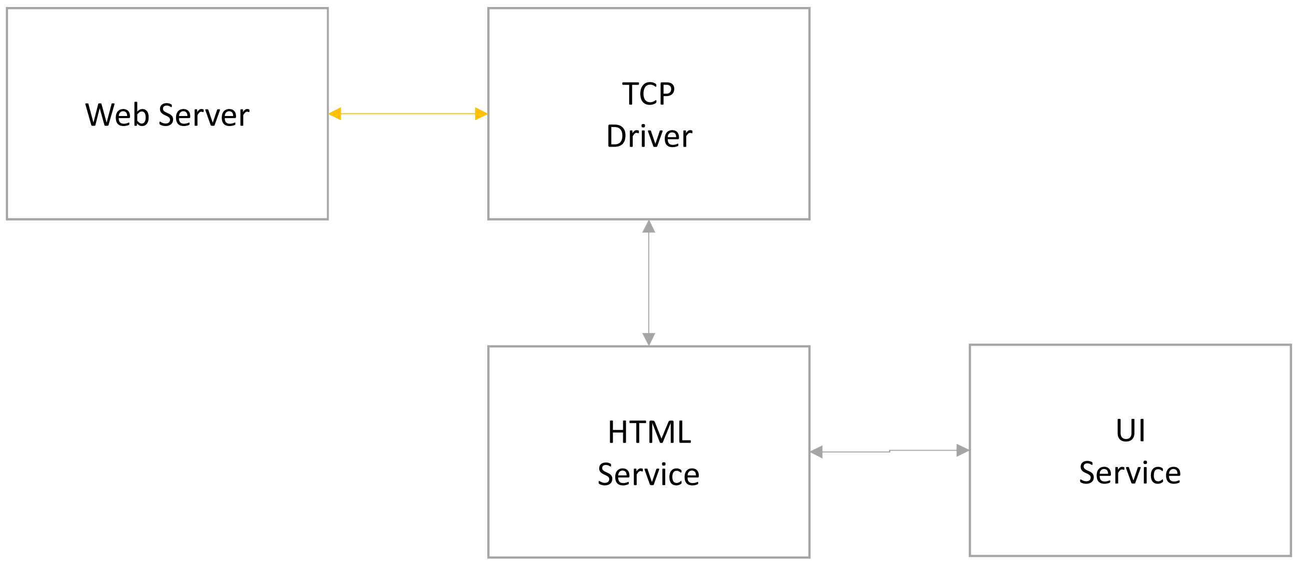
```
/**
 * Table of contents approach
 */
int main(int argc, char** argv) {
   int output = step1();
   step2(output);
}
```

```
/**
 * Interacting services
 */
int main(int argc, char** argv) {
   MyService service1;
   OtherService service2;
   service1.register(service2);
}
```

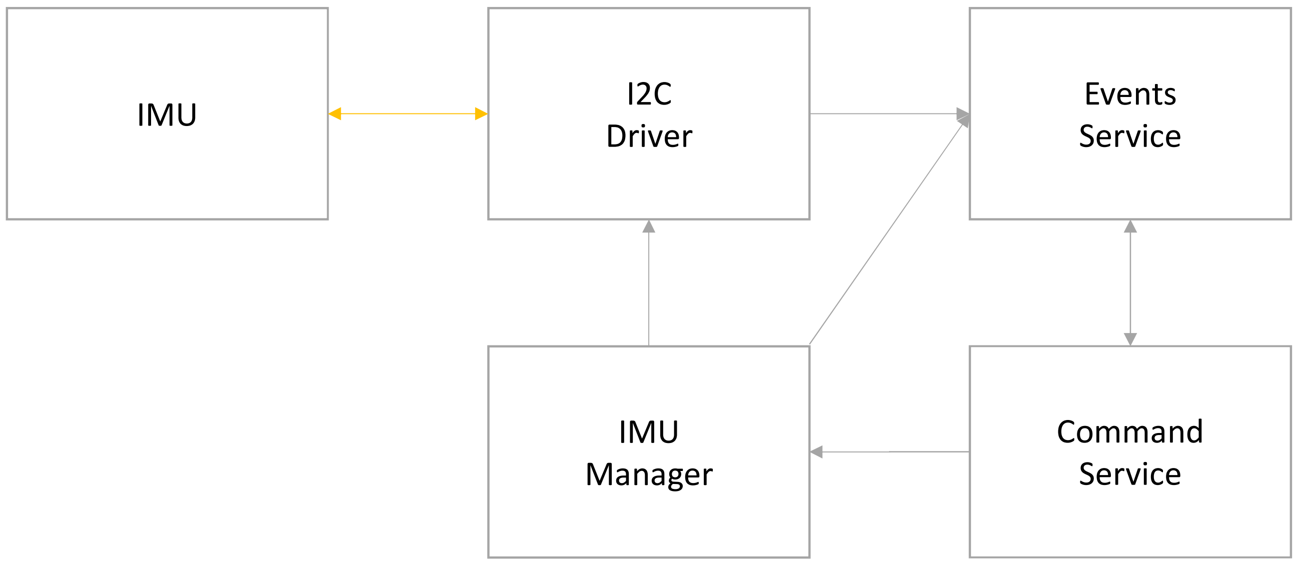
Software as a System: Systems Design



Software as a System: Web Browser Example



Software as a System: Embedded Example



Systems Breakdown and Design

System Design

- Topologies are compositions of components:
  - Set of components
  - Connections between components
- Identify system's components
- Identify connections between components
- Sketch software topology and software component interaction

- Examples:
  - Motor controller -> software motor manager
  - Motor controller -> hardware driver (UART, SPI, I2C…)
  - Radio communication -> radio component manager
  - Radio communication -> radio hardware driver (SPI, etc…)

Interface Design

- Interfaces specify interactions between components:
  - Expose certain functionality
  - Protocol used for communication (Function calls, Register Writes, IPv4)
  - May exchange data
- Identify functionality to be exposed
- Identify communication protocol
- Establish data to be exchanged and ownership of that data

- Examples:
  - Event manager sends F´ packets to radio manager via port call
  - Radio manger sends byte data via a function call to SPI driver
  - SPI driver writes hardware registers to trigger telecom transmission

Component Design

- Components implement functionality of interfaces:
  - Implements and uses set of interfaces
  - Executes within a particular context
  - Produces, consumes, or manages data
- Identify interfaces implemented and used by component
- Select execution context
  - Caller, thread, ISR, etc.
- Determine needed shared data

- Examples:
  - IMU manager has I2C read, and I2C write ports and executes on callers thread
  - TCP driver uses Berkley socket interrace to IP stack on a dedicated read thread

Data and Ownership

- Identify shared data between components
- Establish how shared data is allocated, exchanged, and deallocated
- Designate the owner of shared data items at all times
- Identify off-nominal handling conditions

- Example:
  - Framer component allocates memory via buffer manager, receiving ownership of shared buffer
  - Framer delegates ownership to IPv4 driver via port call
  - IPv4 component delegates ownership back to buffer manager deallocating the shared memory

Off-Nominal Conditions

- Identify places where non-standard conditions can occur
- Identify the severity of the condition
- Identify appropriate response to the condition

- Examples:
  - Hardware failure -> go to safe mode
  - Malformed user input or data -> emit warning event
  - Memory inconsistency -> full system reset

Flight Software Design Considerations

Startup: Initialization and Allocation

- Finite resources are allocated at initialization to reduce risk
- Typical resources include: RAM, Threads, Critical Files
- Dynamic resources draw from preallocated pool; failures handled

- Implications:
  - Static memory or initialization allocated heap
  - Preallocated worker threads
  - Preallocated critical files
  - No recursion
  - Buffer managers, file managers handle unpredictable requests

Synchronous Execution, Concurrency, Threads

- Synchronous execution uses caller's execution context
- Parallel execution has multiple execution contexts via threads
- Sharing data across contexts requires locking and/or queues
- "Ships passing in the night"

- Implications:
  - Must plan concurrency model
  - Shared resources must be handled appropriately
  - Messaging and scheduling must be thought through
  - Care must be taken with work requiring specific timing

Deadlines, Timeliness

- Some processes have strict timing or deadlines
- Identify tasks that require strict timing, active processing, and background processing
- Strict timing needs specific deadline handling

- Implications:
  - Non-time sensitive work should be placed in low-priority background tasks
  - Critical work without deadlines occupies medium priority tasks
  - Work with specific deadlines goes in the highest priority tasks

Faults, FATALs, and Error Handling

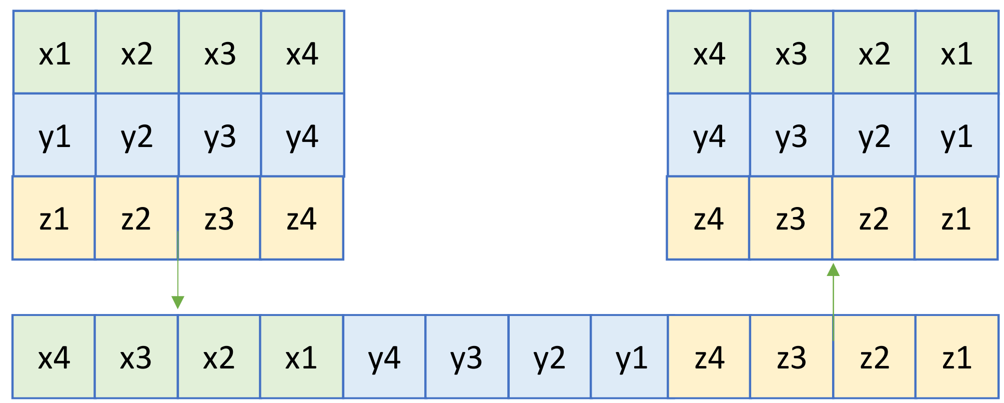
- Flight software is expected to protect the spacecraft
- Off-nominal conditions will always occur; the universe desires this
- Spacecraft operators need to understand cause of behavior

- Implications:
  - Uncontrolled reboots and crashes should be avoided
  - Logging of off-nominal conditions should occur
  - Spacecraft should be made safe before loss-of-control responses

Data Serialization and Deserialization

- Data in RAM may be padded, expanded, or mixed with other values
- Bytes in RAM may have different orders between different machines
- Data in transit should be an array of bytes in specified order

- Implications:
  - Data exchange format must be well-specified and obeyed
  - Data cannot always be directly copied into buffers



Modeling Flight Software in F´

Topologies

- Topologies represent a network of components
- Contain instantiations of each component
- List connections between the ports of all the components

```
connections Downlink {
  chanTlm.PktSend -> comQueue.comQueueIn[0]
  eventLogger.PktSend -> comQueue.comQueueIn[1]

  fileDownlink.bufferSendOut -> comQueue.buffQueueIn[0]
  framer.bufferDeallocate -> fileDownlink.bufferReturn

  comQueue.comQueueSend -> framer.comIn
  comQueue.buffQueueSend -> framer.bufferIn

  framer.framedAllocate -> comBufferManager.bufferGetCallee
  framer.framedOut -> radio.comDataIn
  comDriver.deallocate -> comBufferManager.bufferSendIn

  radio.drvDataOut -> comDriver.send
  comDriver.ready -> radio.drvConnected
  radio.comStatus -> comQueue.comStatusIn
}

connections FaultProtection {
  eventLogger.FatalAnnounce -> fatalHandler.FatalReceive
}
```

Ports

- Represent interface to components
- Form a point-to-point network for communication
- Have arguments with specific types
- May have return values

```
struct ImuData {
  $time: Fw.Time
  vector: Vector
  status: Svc.MeasurementStatus
} default { status = Svc.MeasurementStatus.STALE }

@ Port for receiving current X, Y, Z position
port ImuDataPort() -> ImuData
```

Components

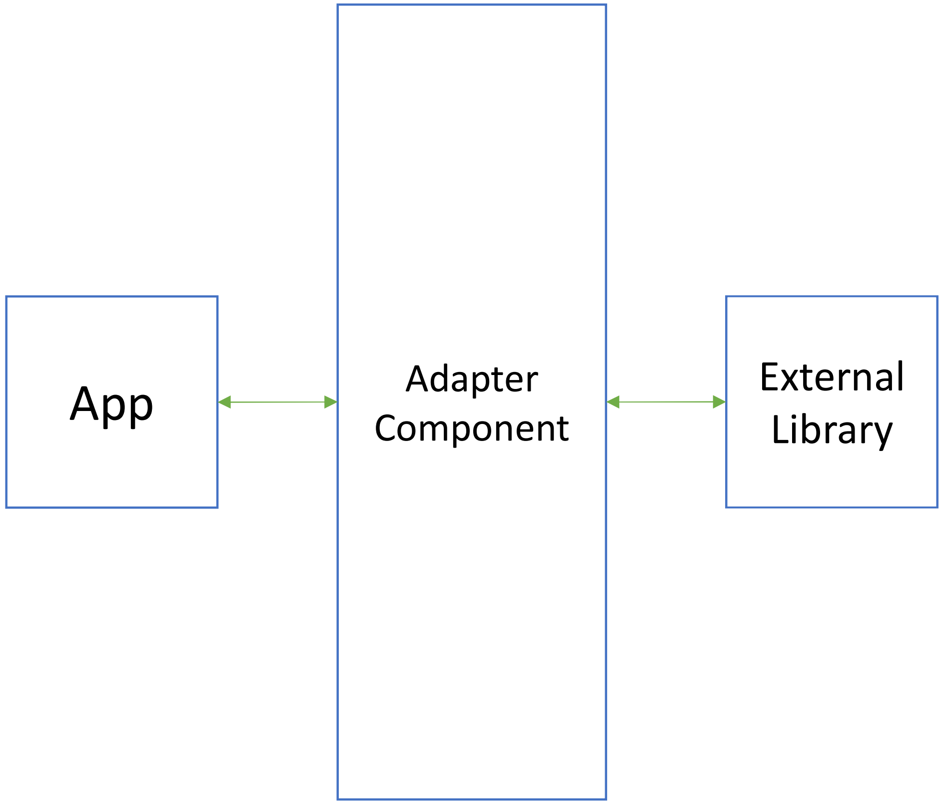
- Represent concrete function in the system
- Come in three variants: Active, Passive, and Queued relating to execution context
- Communicates with other components via ports

```
module Gnc {
    @ The power state enumeration
    enum PowerState {OFF, ON}

    @ Component for receiving IMU data via poll method
    passive component Imu {
        @ Port to send telemetry to ground
        guarded input port Run: Svc.Sched


        @ Command to turn on the device
        guarded command PowerSwitch(
            powerState: PowerState
        ) \
        opcode 0x01


        @ Event where error occurred when requesting telemetry
        event TelemetryError(
            status: Drv.I2cStatus @< the status value returned
        ) \
        severity warning high \
        format "Telemetry request failed with status {}" \

        @ X, Y, Z degrees from gyroscope
        telemetry gyroscope: Vector id 1 update always format "{} deg/s"

    }
}
```

F´ Design Patterns

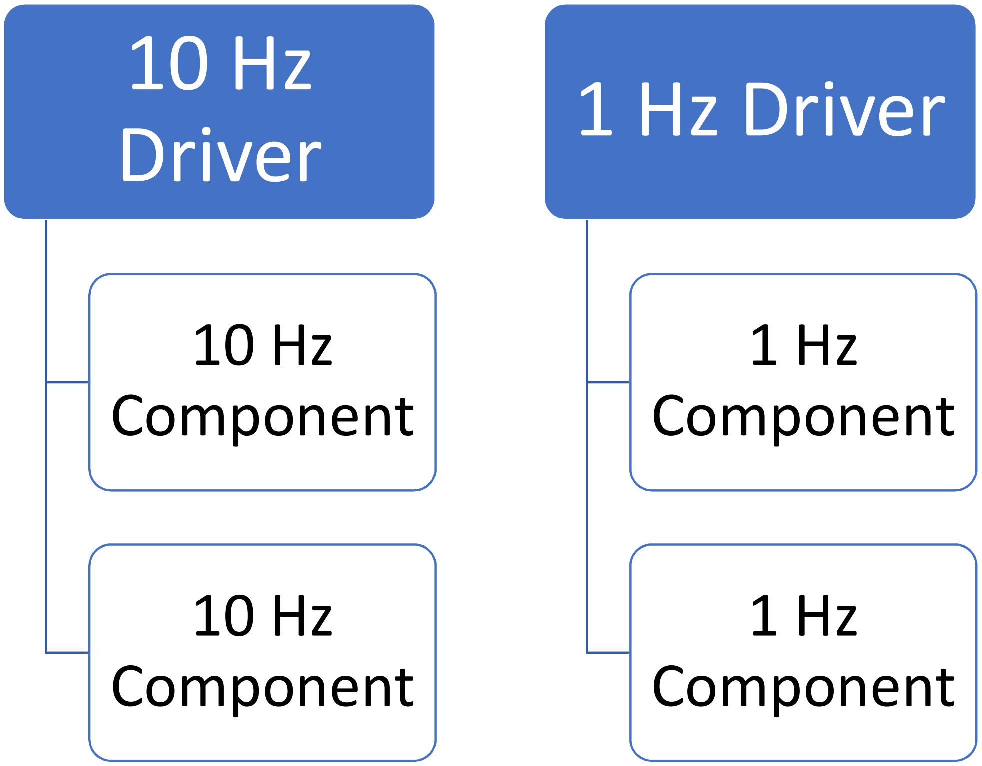
Adapter Pattern

- Adapts “something else” to work within F´
- Typically done by writing an F´ component bridging functionality
- Adapts or adds concurrency and timeliness considerations
- Adds commands, events, and telemetry



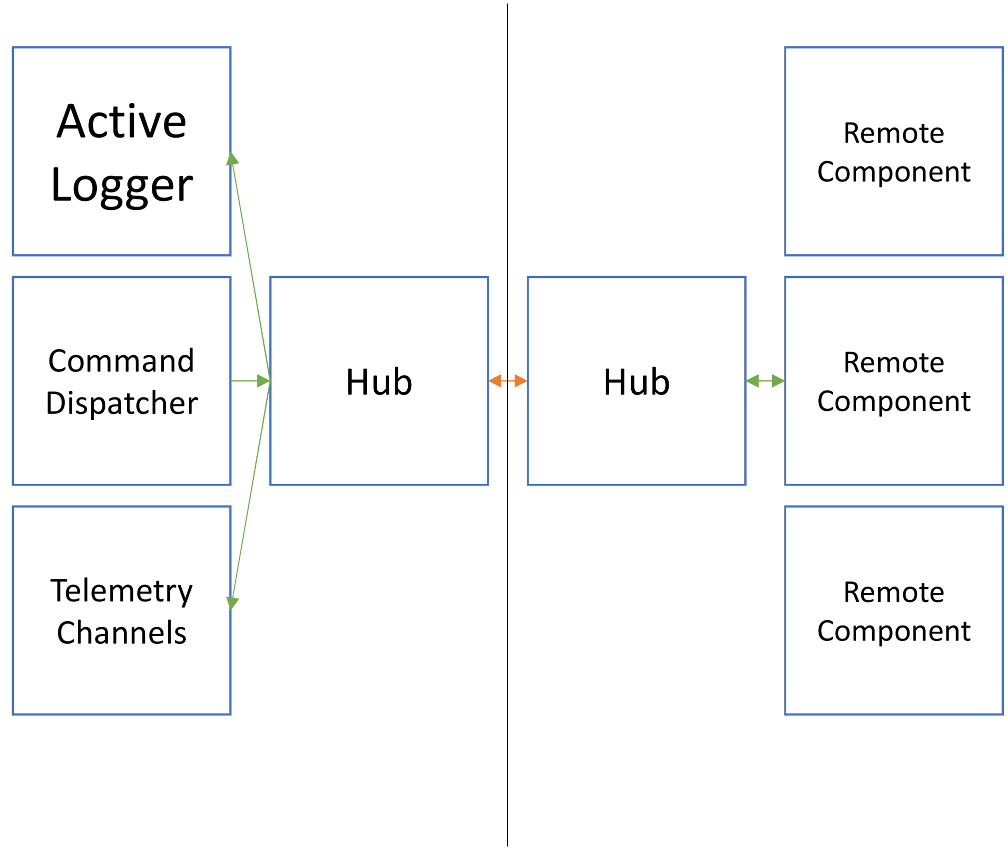
Rate Group Pattern

- Drives components at a set rate
- Simple provider of timeliness, allowing work at set time
- Care must be taken with other forms of concurrency
- Care must be taken to not slip



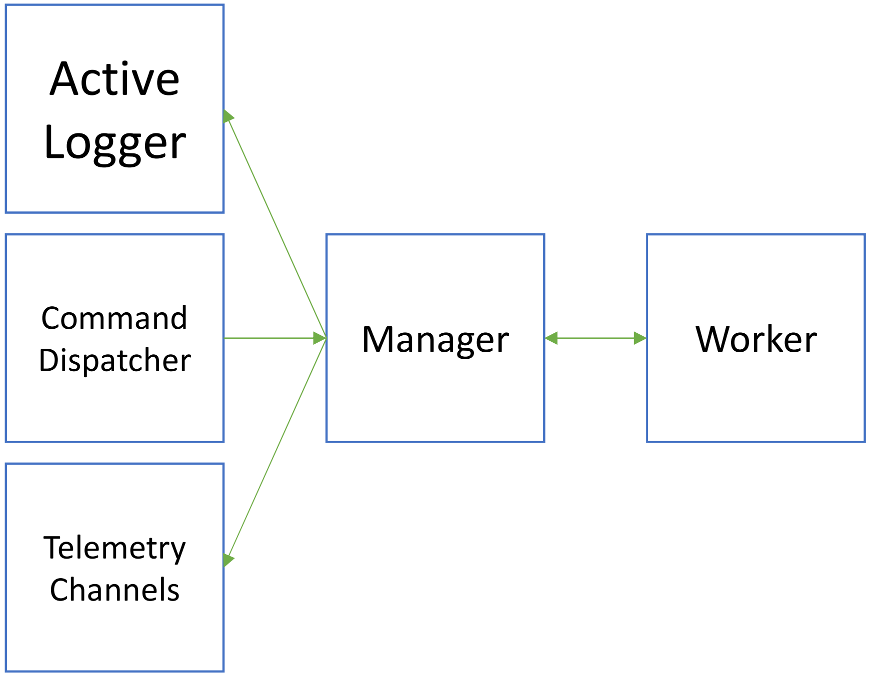
Hub Pattern

- Routes multiple F´ ports across some communication layer
- Unwraps on the far side of the communication layer
- Allows for inter-process communication



Manager-Worker Pattern

- Decouples long-running tasks from need for quick interaction
- Manager sends work to worker worker responds back afterwards
- **Only** Manager communicates with worker
- Parallels “worker thread” pattern



Jet Propulsion Laboratory
California Institute of Technology

jpl.nasa.gov