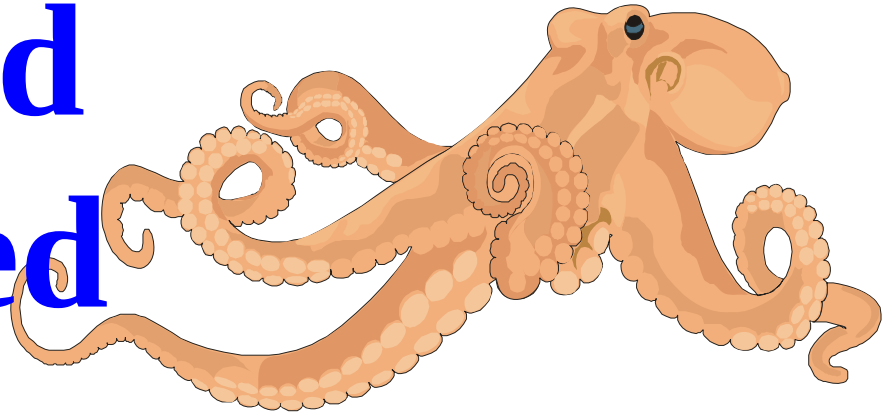
7

Distributed +Embedded Systems



Distributed Embedded Systems

Philip Koopman

September 16, 2015

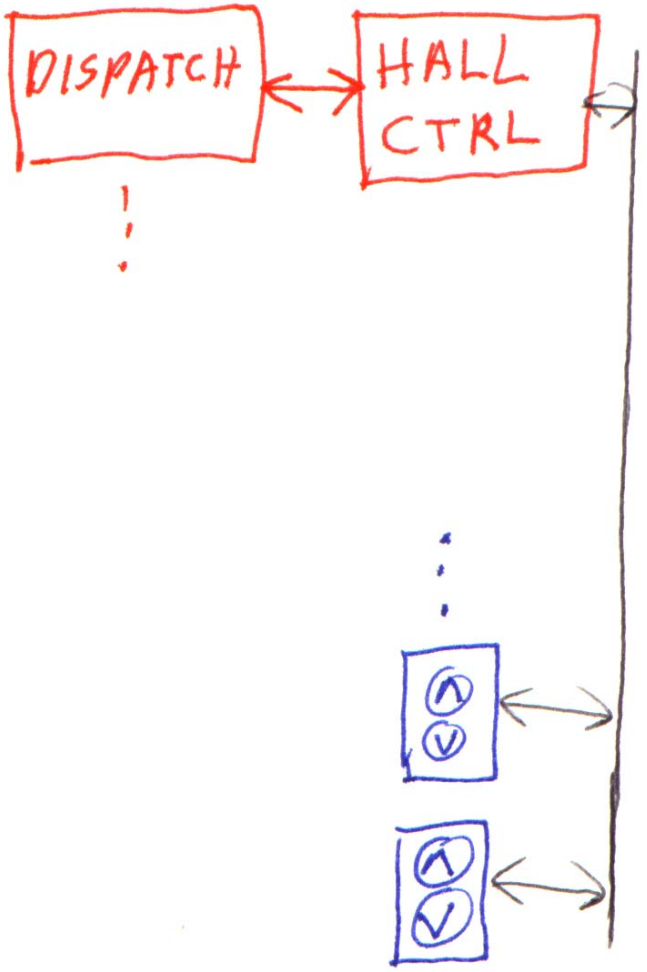
Carnegie Mellon

Introduction

- **Time Trigger vs. Event Trigger**
  - Two different approaches to networking
- **Distributed embedded systems – progression of ideas**
  - Distributed power switching
  - Muxed control signals
  - Distributed computation
  - “Smart nodes” – an extreme case of distribution
- **Distributed vs. centralized tradeoffs**
  - List of common wins, loses, and draws
  - In industry, tradeoff studies often try to justify things based on the “draws” instead of the “wins”

Example System – Elevator Hall Call Button

- **Up Button**
  - Button press
- **Up Button Light**
  - Light on
  - Light off
- **Down Button**
  - Button press
- **Down Button Light**
  - Light on
  - Light off



Elevator Call Button: Event-Based Messages

- **Button Press**
  - Turn on button light
  - Send a “button pressed” message every time the button is pressed
  - Keep button light turned on for 500 msec after button pressed
    - Then turn button light off *unless* an acknowledgement message is received
  - Turn button light off when told to by hallway computer

- **Hallway computer (one for all floors)**
  - When button is pressed, send acknowledgement to keep light on
  - When button is pressed, notify dispatching computer to send elevator car
  - When car doors open, send “turn button light off” message

- **For dependable systems, need handshake messages at every step**
  - Might be at application layer
  - Might be at network protocol layer

Elevator Call Button: Time-Triggered Messages

- **Send "button" and "button light" messages every 150 msec**
  - Each message has current state of "on" or "off"
- **Button Press**
  - Set "button state" variable to "pressed" (gets copied to hallway computer)
  - Turn on button light and force on for 500 msec; then follow message value
  - Whenever you can, check the button light state and set light accordingly (perhaps every 150 msec)
- **Hallway computer (one for all floors)**
  - While button is pressed, notify dispatching computer to send elevator car
    - Set button light state to "on"
  - When car doors open, set button light state to "off" & button state to "off"
- **Message acknowledgements often not required**
  - Missed button light state messages are self-correcting
  - User must correct a completely missed "button press" message by pressing again

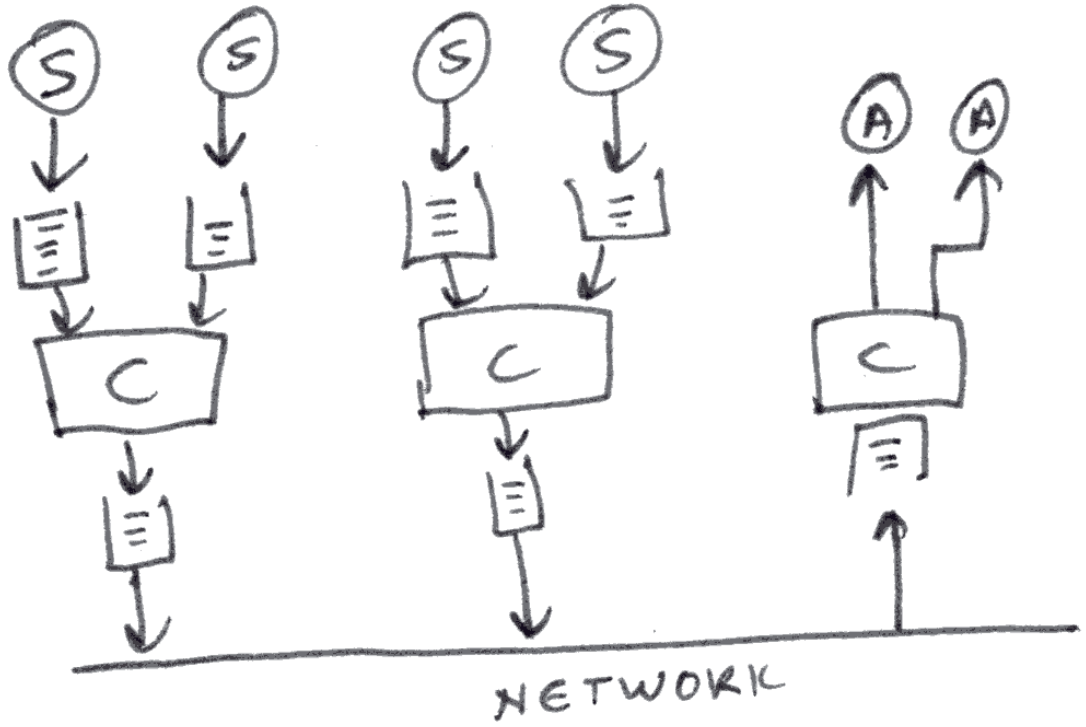
Tradeoff: When Do You Send A Message?

- **Event-based messages** **(“events”)**
  - Send messages in response to an event
    - Similar to event-triggered systems
    - Think “asynchronous” state machine transitions
  - Can have high message frequency with an “event shower”

- **Time-Triggered messages** **(“state variables”)**
  - Message sending is invisible to application programmer
    - Send messages periodically with the latest information
    - Think “synchronous” state machine transitions
    - “Looks” like a very small piece of shared memory
  - Reduces problems with missed messages
  - Matches communication & computing load to system control loop speeds rather than external environment
  - Have to infer that an event happened based on observing a value change
    - Might be easy to miss a sequence of quick events

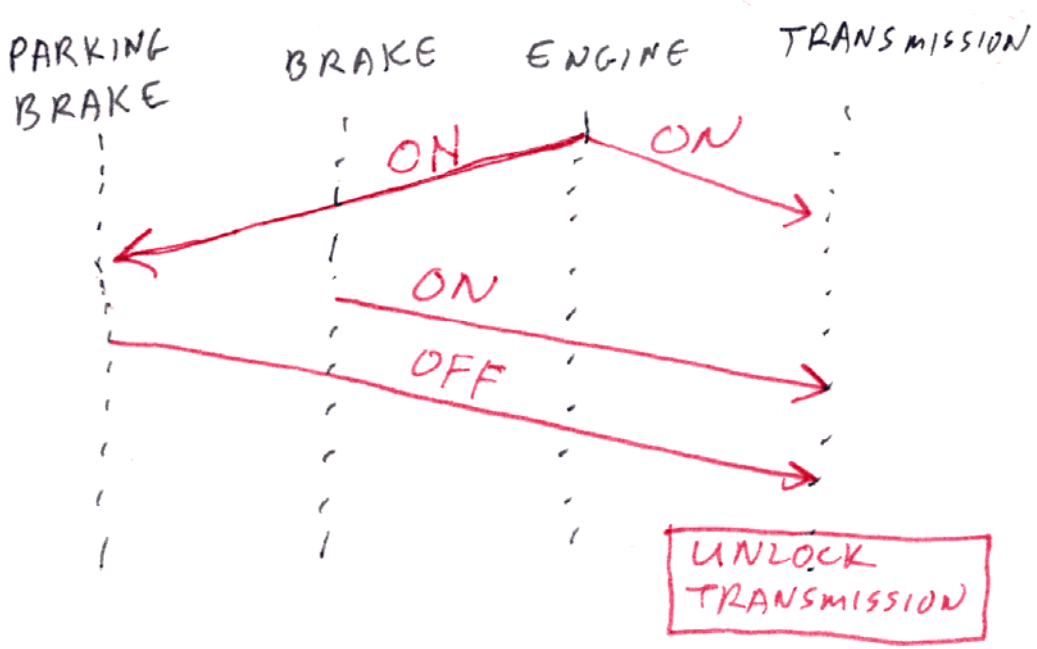
Generic Event-Triggered Approach

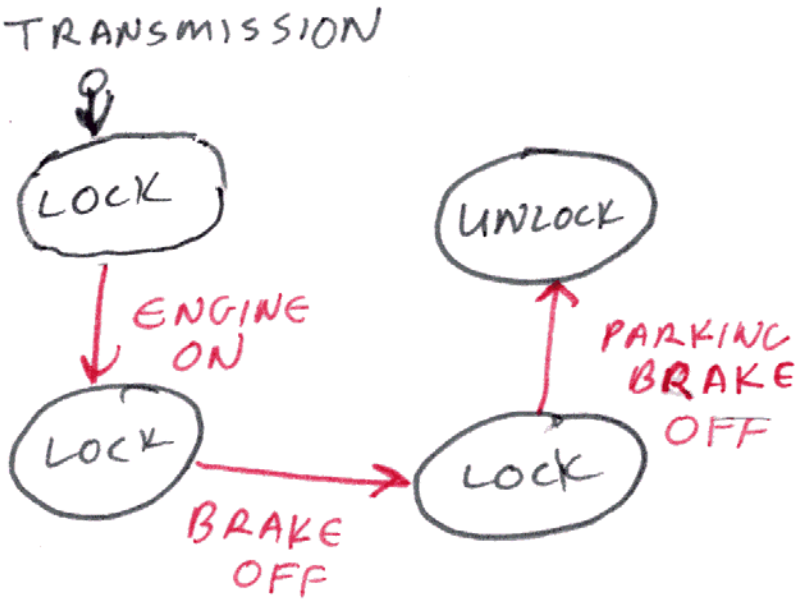
- **Sensors feed computers**
  - But their values arrive asynchronously – whenever values are available
  - So, sensor values must be queued
- **Computers are demand driven**
  - They process values from queues when available
  - Network messages are queued for transmission
- **Network is demand driven**
  - Transmission queues release messages according to some network protocol
- **Actuators are demand driven**
  - Actuator outputs occur in response to event messages



Problem – Coordinating Events

- **Car example: transmission unlocks to shift out of park if:**
  - Engine is running AND
  - Parking brake is OFF AND
  - Brake is ON

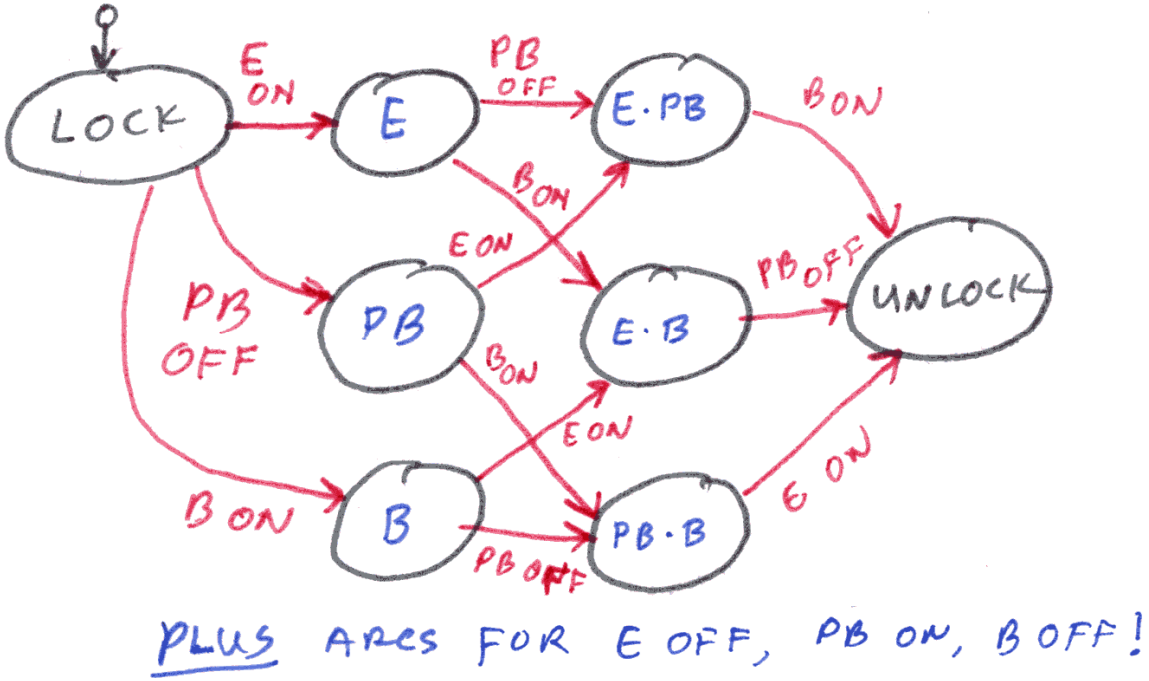




- **But, this is *too simplistic*!**
  - There are many arrival orders

Asynchronous State Machine Problem

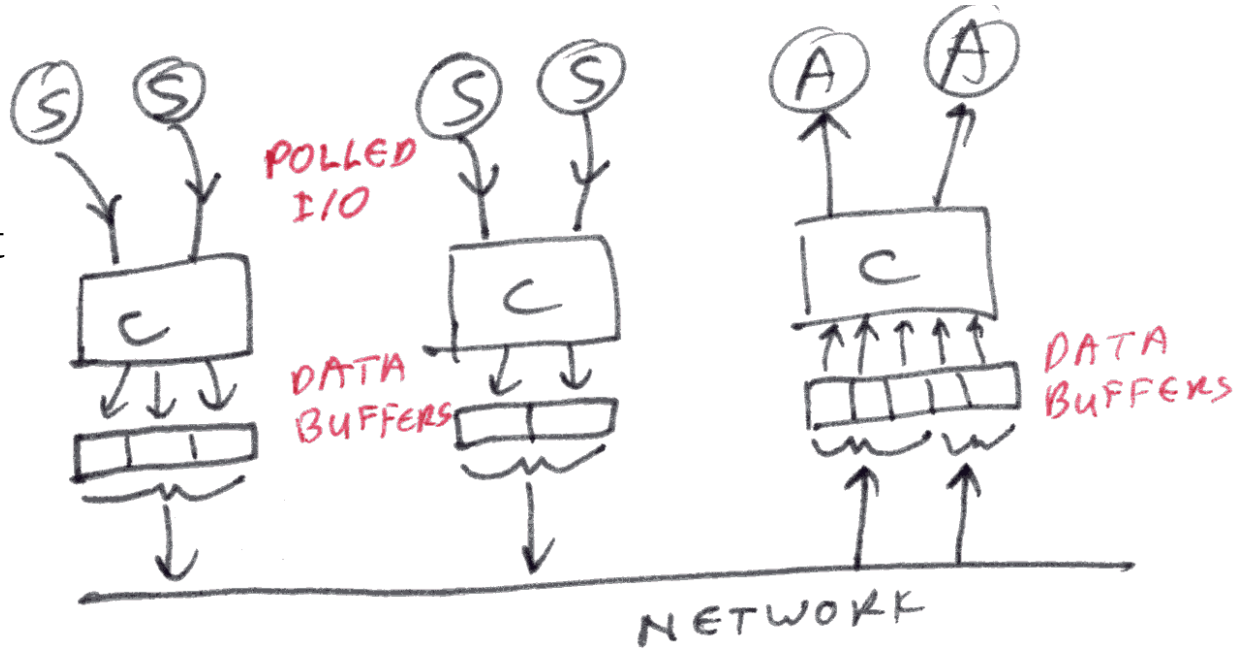
- **Asynchronous state machines have trouble with complex condition**
  - Need to use states to keep track of which events have been seen if order is flexible
  - (This is also a problem with UML sequence diagrams)



- **Other models are possible that work around this**
  - Colored Petri nets handle this situation better
  - But they can have problems with interrupts and asynchronous events

Generic Time-Triggered Approach

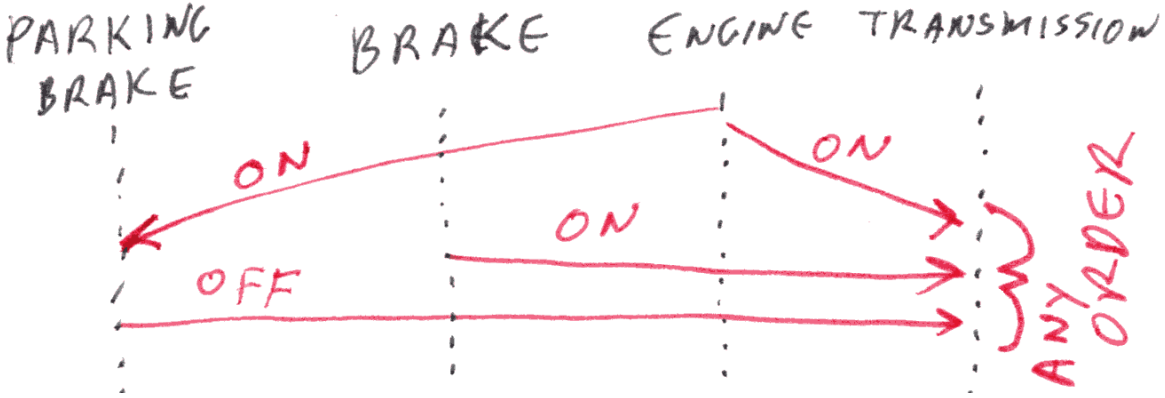
- **Computers poll sensors**
  - Sensors queried periodically
  - Sampled whether or not there has been a change
- **Computers are time triggered**
  - Sensors are sampled when it is time perform a computation
  - State variables are placed in buffers for later transmission on network

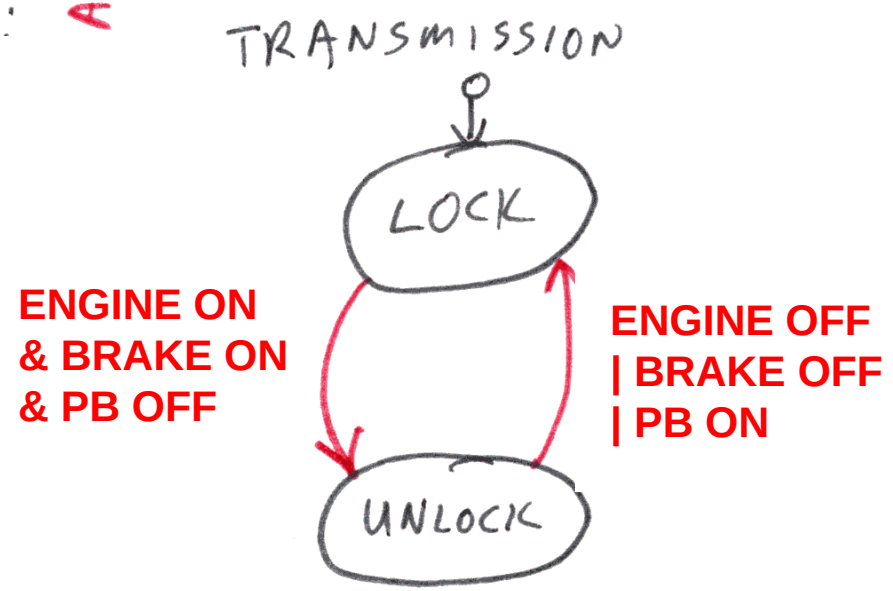


- **Network is periodic**
  - Copies of buffers are sent to all nodes periodically
  - Receiving nodes store most recent values, even if previous value unused
  - Remember the "blackboard architecture?" The receive buffers are that node's locally stored copy of the current blackboard contents
- **Actuators are periodically driven**
  - Actuators outputs asserted periodically regardless of change/no change

Time Triggering Simplifies Event Coordination

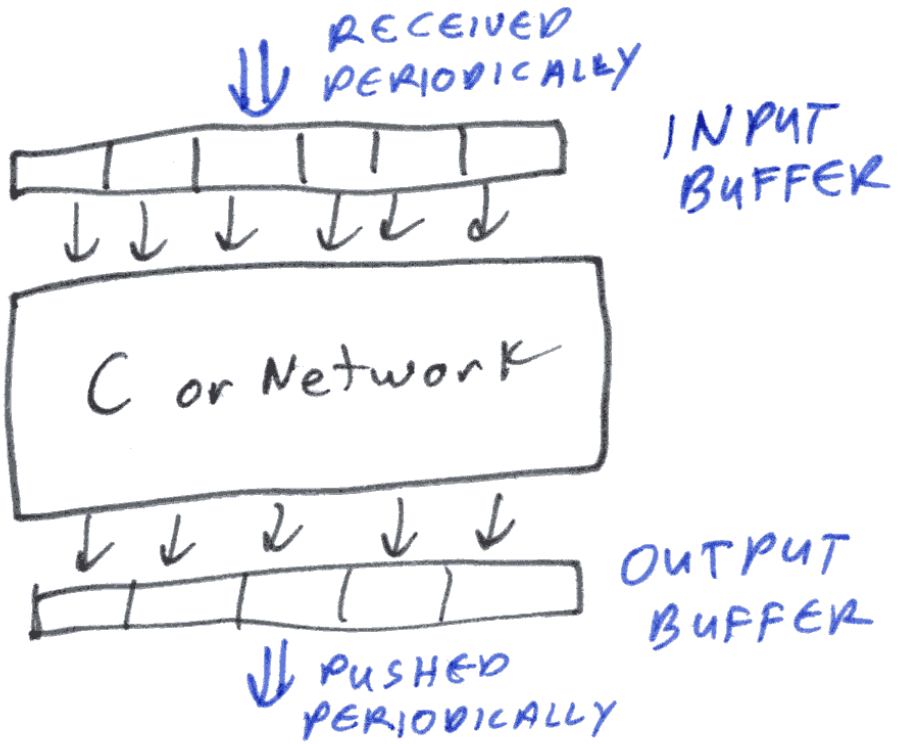
- **Example: transmission unlocks to shift out of park if:**
  - Engine is running AND
  - Parking brake is OFF AND
  - Brake is ON





General Time Triggered Design Notes

- **Design based on periodically updated state variables**
  - Every compute/communication block has a state variable buffer
  - Buffer contents are updated whenever source sends an update
  - Buffer values are read whenever consuming functional block needs them
  - Generally, each value is updated atomically, but related values might not be updated together (so they might be out of synch by a cycle)



Design Difference: Event vs. Time Triggered

- **Event triggered:**
  - State machine only changes states when event occurs
    - Actions within state are executed exactly once, when state is entered
  - Each arc can have ONLY ONE incoming event/message
    - Instantaneous state changes
    - Events arrive via network message or are serialized in some other way
    - Requirements statements can depend on exactly ONE message (and possibly state variables specifically managed by the object)
  - In many cases these are simpler to understand & design
    - But, they are easily confused by dropped, duplicated, or out-of-order messages
    - Coordinating multiple messages requires messy temporary variables

- **Time triggered:**
  - State machine changes periodically based on variables and most recent inputs
    - State machine has to infer events by noting state changes
    - Can change state based on multiple messages and/or local variable values
    - Actions within each state are executed continually
  - Tend to be more robust to dropped or out-of-order messages

Mindless Formula for TT Behavioral Requirements

- **Time-triggered system:**
  - ***<ID> (Null or <message value>, … <message value>)***

    and *(Null or <variable value test>, … <variable value test>)*

    shall result in *<message transmitted>, …*

    <variable value assigned>, …
  - **Can trigger on zero or more messages; zero or more variables**
    - OK for left hand side trigger to ONLY be a state variable (or always be true)
    - Right hand side can have zero or more messages; zero or more variable values
    - "Shall" or "should" are both acceptable
  - OK to transmit multiple messages; OK to assign multiple variables
  - EVERY VERB GETS A NUMBER

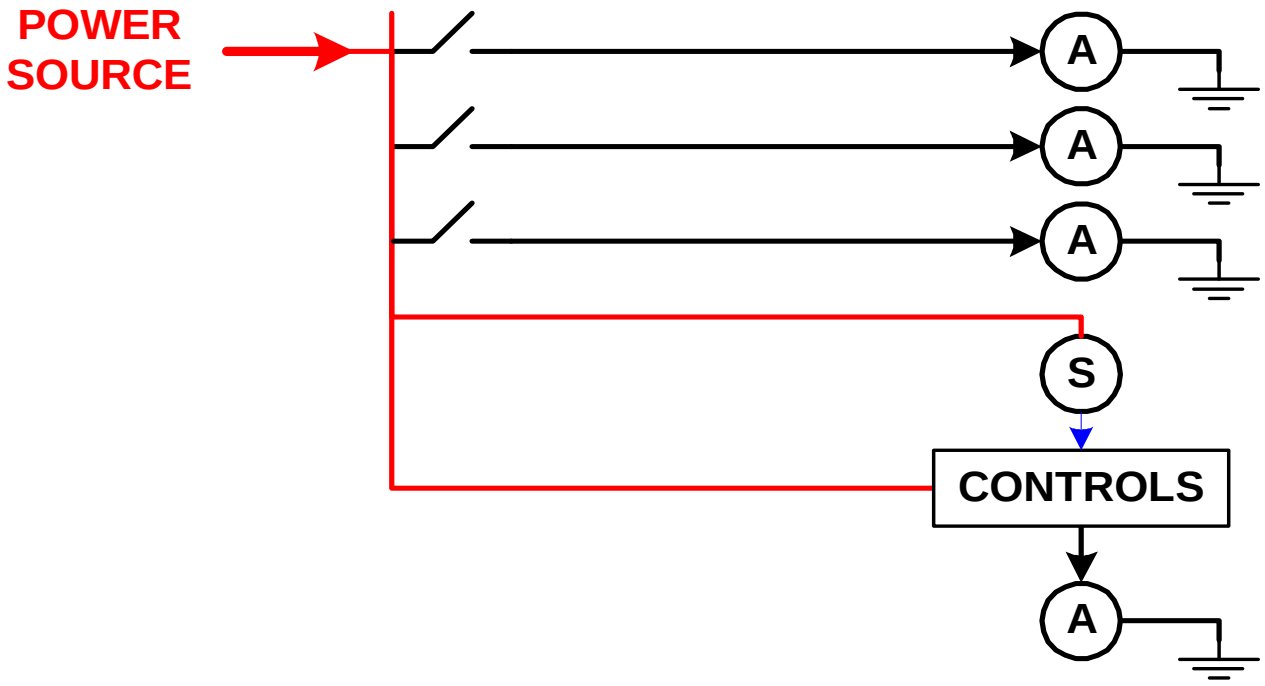
Questions on TT vs. ET?

Distributed Computing – Step By Step Evolution

Let's work through how systems evolve from electromechanical to distributed

STEP 1 – no computer at all

- **You don't need a computer at all to operate light bulbs with switches!**
- **Non-computer control is possible**
  - RLC circuits, analog transistors, etc.



Why Add A Computer?

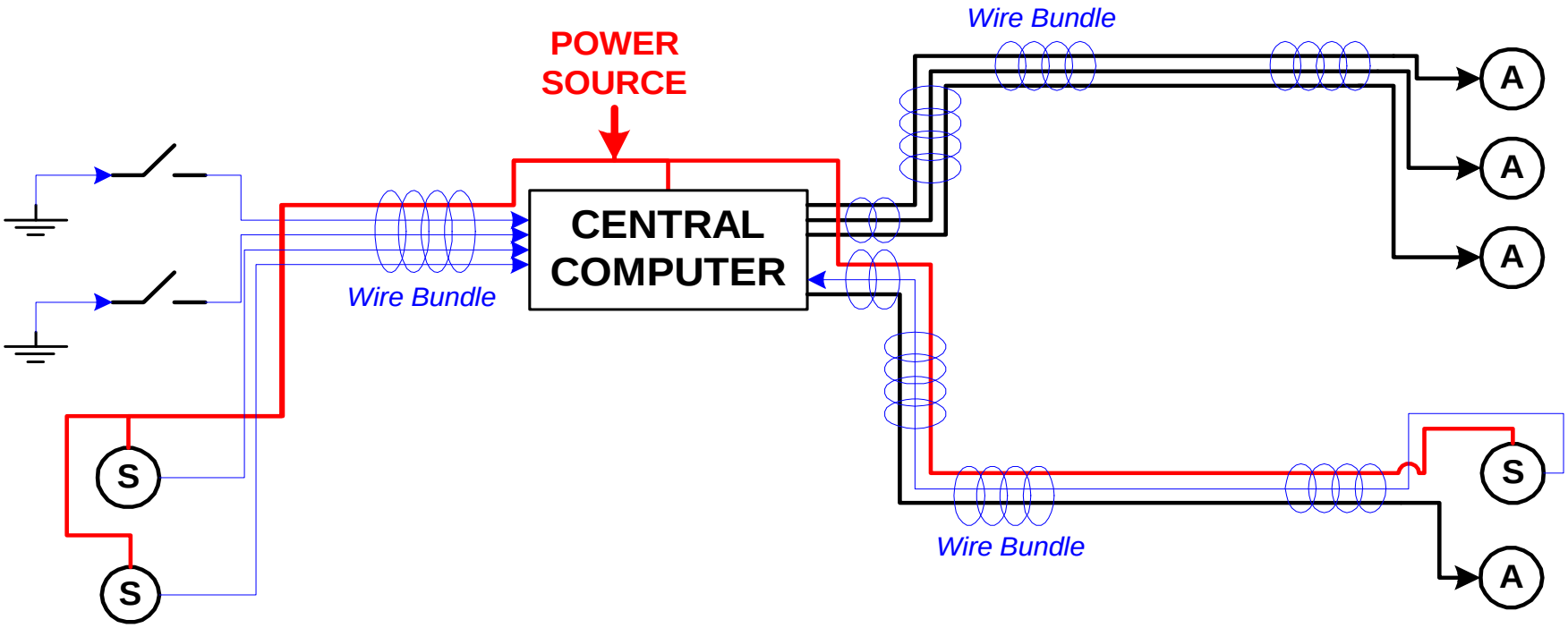
- **Permits optimization**
  - Using a many-to-many relationship for actuators to control sensors requires relays or other signal isolation (isolation relays would also be “computation”)
  - Can change control loop behavior depending on system operating modes

- **Enables adding sophisticated features in software**
  - Safety interlocks
    - Require brake pedal to be depressed before shifting into gear
  - Active safety operations
    - Car locks doors once shifted into gear
  - Timers, counters, conditional logic
    - Variable interval wipers that adapt to rain intensity

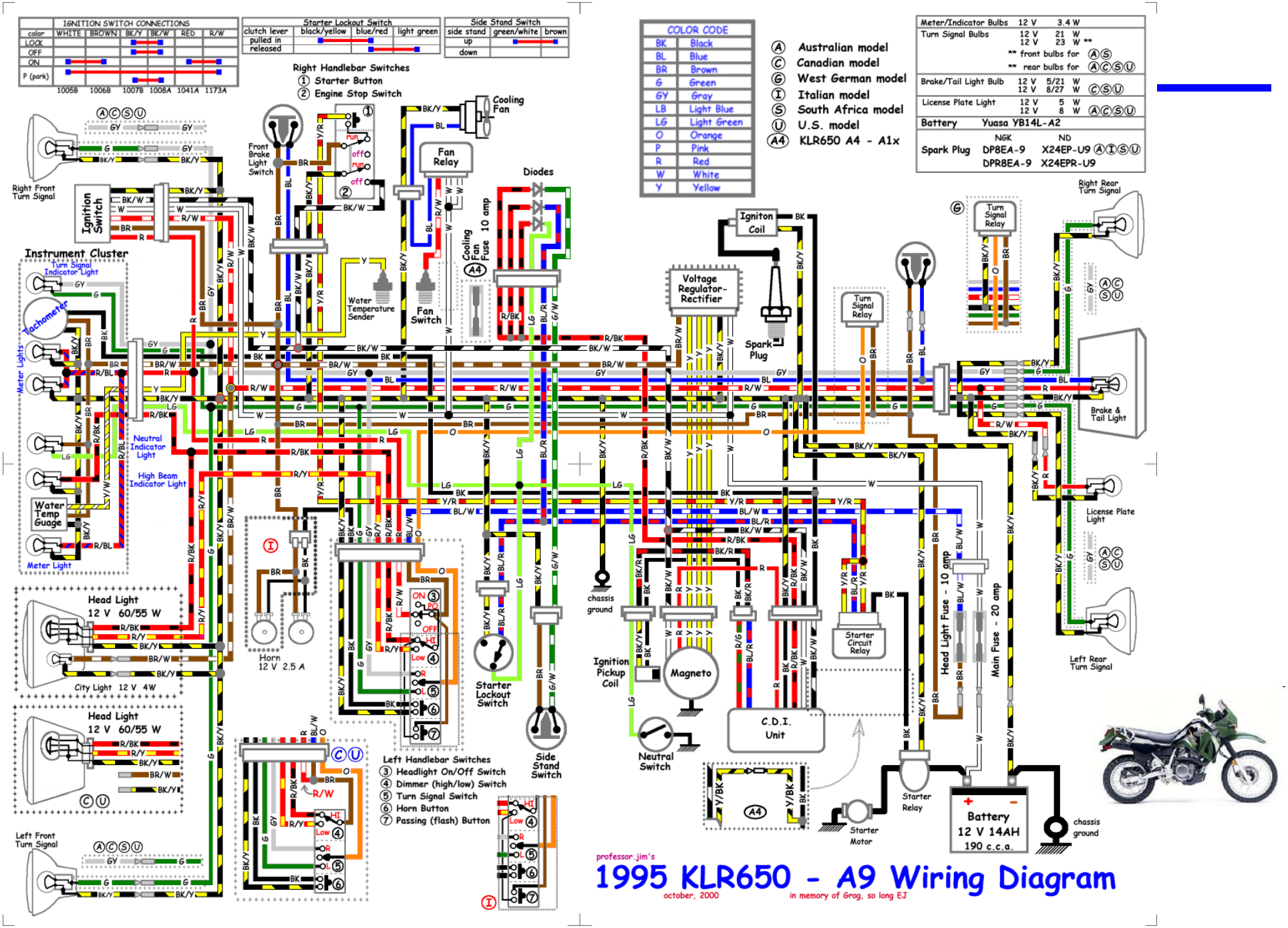
- **Enables context-sensitive operation**
  - Switches can be momentary closure “soft switches”
  - Permits controller to set value even when switch isn’t pressed
    - Example: rear view window heaters off after timeout or if outside temp is hot

Centralized System

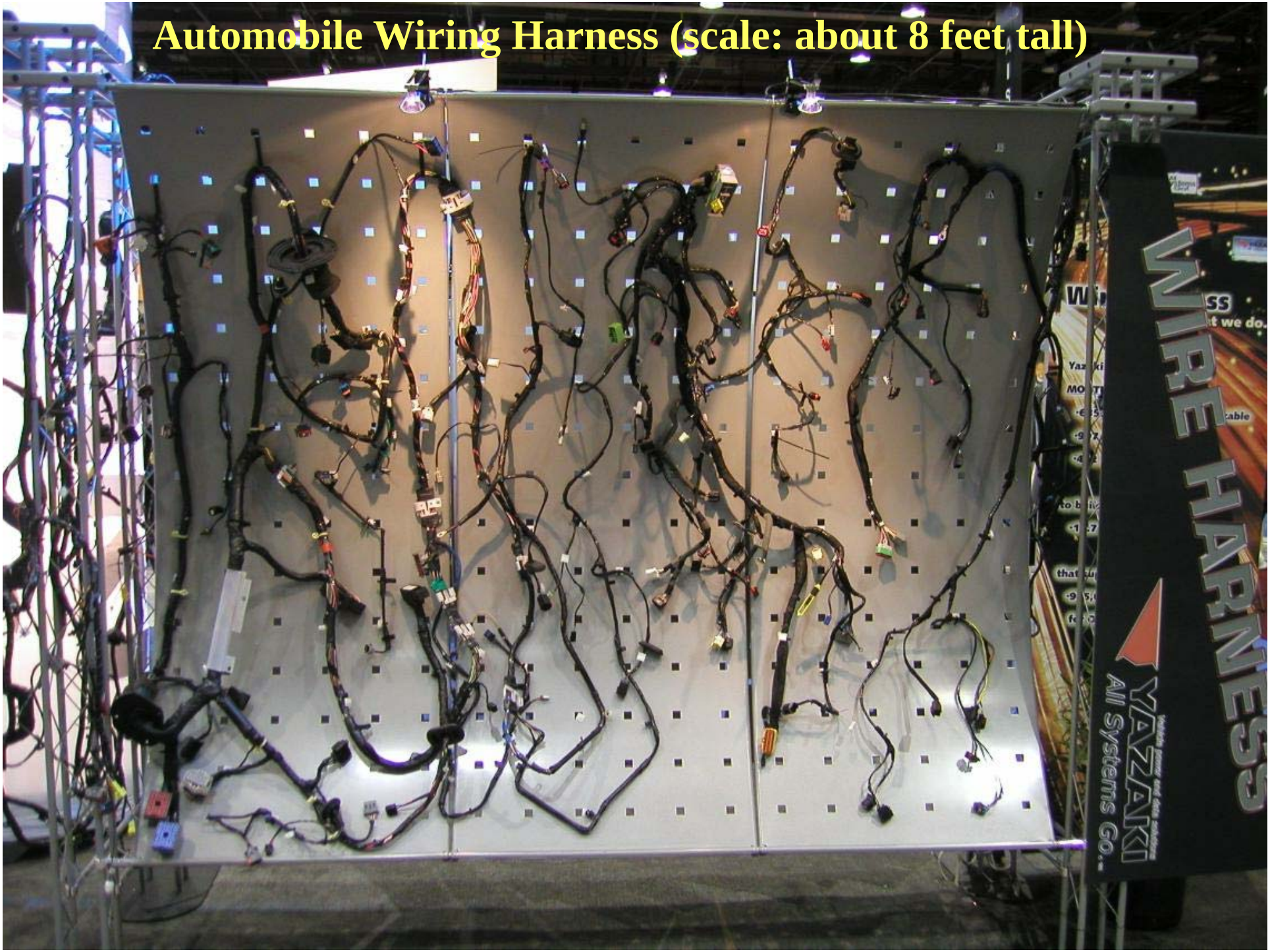
- **Central computer**
  - Reads input sensors
  - Provides motive power to actuators
  - Common ground (e.g., metal vehicle frame) used as electrical return path



http://www.baesystems.com/gallery/air/images/FADEC_Full_Authority_Digital_Engine_Control_Testinghires.jpg


IGNITION SWITCH CONNECTIONS
color | WHITE | BROWN | BK/Y | BK/W | RED | R/W
LOCK
OFF
ON
P (park)
1005B 1006B 1007B 1008A 1041A 1173A
Starter Lockout Switch
clutch lever | black/yellow | blue/red | light green
pulled in
released
Side Stand Switch
side stand | green/white | brown
up
down
COLOR CODE
BK Black
BL Blue
BR Brown
G Green
GY Gray
LB Light Blue
LG Light Green
O Orange
P Pink
R Red
W White
Y Yellow
Ⓐ Australian model
Ⓒ Canadian model
Ⓖ West German model
Ⓘ Italian model
Ⓢ South Africa model
Ⓤ U.S. model
Ⓐ4 KLR650 A4 - A1x
Meter/Indicator Bulbs 12 V 3.4 W
Turn Signal Bulbs 12 V 21 W ** 12 V 23 W **
** front bulbs for ⒶⓈ
** rear bulbs for ⒶⒸⓈⓊ
Brake/Tail Light Bulb 12 V 5/21 W 12 V 8/27 W ⒸⓈⓊ
License Plate Light 12 V 5 W 12 V 8 W ⒶⒸⓈⓊ
Battery Yuasa YB14L-A2
Spark Plug NGK DP8EA-9 DPR8EA-9 ND X24EP-U9 ⒶⒾⓈⓊ X24EPR-U9
Right Handlebar Switches
① Starter Button
② Engine Stop Switch
Left Handlebar Switches
③ Headlight On/Off Switch
④ Dimmer (high/low) Switch
⑤ Turn Signal Switch
⑥ Horn Button
⑦ Passing (flash) Button
Instrument Cluster
Turn Signal Indicator Light
Tachometer
Meter Lights
Water Temp Guage
Meter Light
Neutral Indicator Light
High Beam Indicator Light
Ignition Switch
Right Front Turn Signal
Left Front Turn Signal
Front Brake Light Switch
Water Temperature Sender
Fan Relay
Fan Switch
Cooling Fan
Cooling Fan Fuse 10 amp
Diodes
Voltage Regulator-Rectifier
Igniton Coil
Spark Plug
Turn Signal Relay
Right Rear Turn Signal
Brake & Tail Light
License Plate Light
Left Rear Turn Signal
Head Light 12 V 60/55 W
City Light 12 V 4W
Horn 12 V 2.5 A
Starter Lockout Switch
Side Stand Switch
chassis ground
Ignition Pickup Coil
Magneto
Neutral Switch
C.D.I. Unit
Starter Circuit Relay
Starter Relay
Starter Motor
Head Light Fuse - 10 amp
Main Fuse - 20 amp
Battery 12 V 14AH 190 c.c.a.
professor jim's
1995 KLR650 - A9 Wiring Diagram
october, 2000
in memory of Grog, so long EJ

Automobile Wiring Harness (scale: about 8 feet tall)



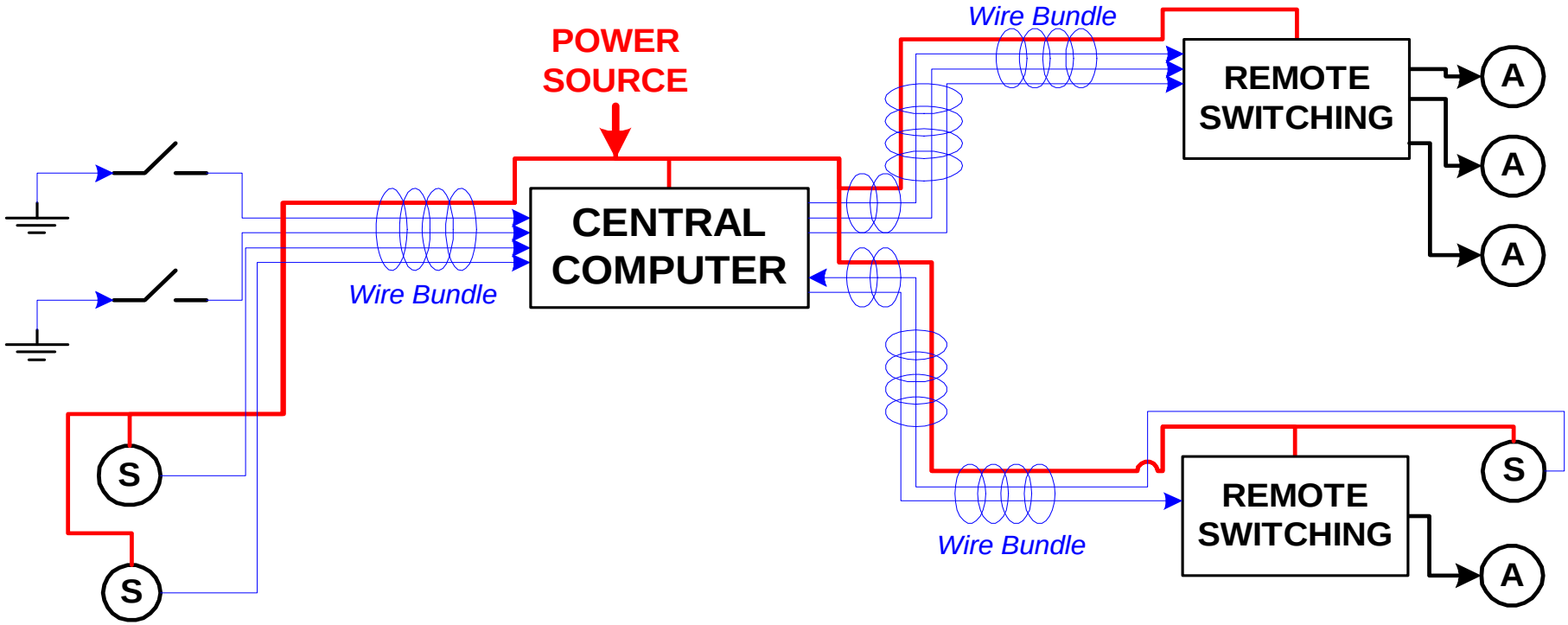
What Can Be Improved Next?

- **Computer has to switch power to actuators**
  - Power switching components can be much larger than microcontroller
  - Power-carrying wires can form a thick, heavy wire bundle – every conductor has to carry activation power, not just a signal

- **So, let's move power switching out to periphery of system**

Remote Power Switched System

- **Actuator outputs from computer are control signals**
  - Power is fed to remote switching modules that obey control signals from CPU
  - No computational power in power switches – can be just power transistors
  - Blue lines are low-current logic signals, not high-current switched power



Why Use Remote Power Switching?

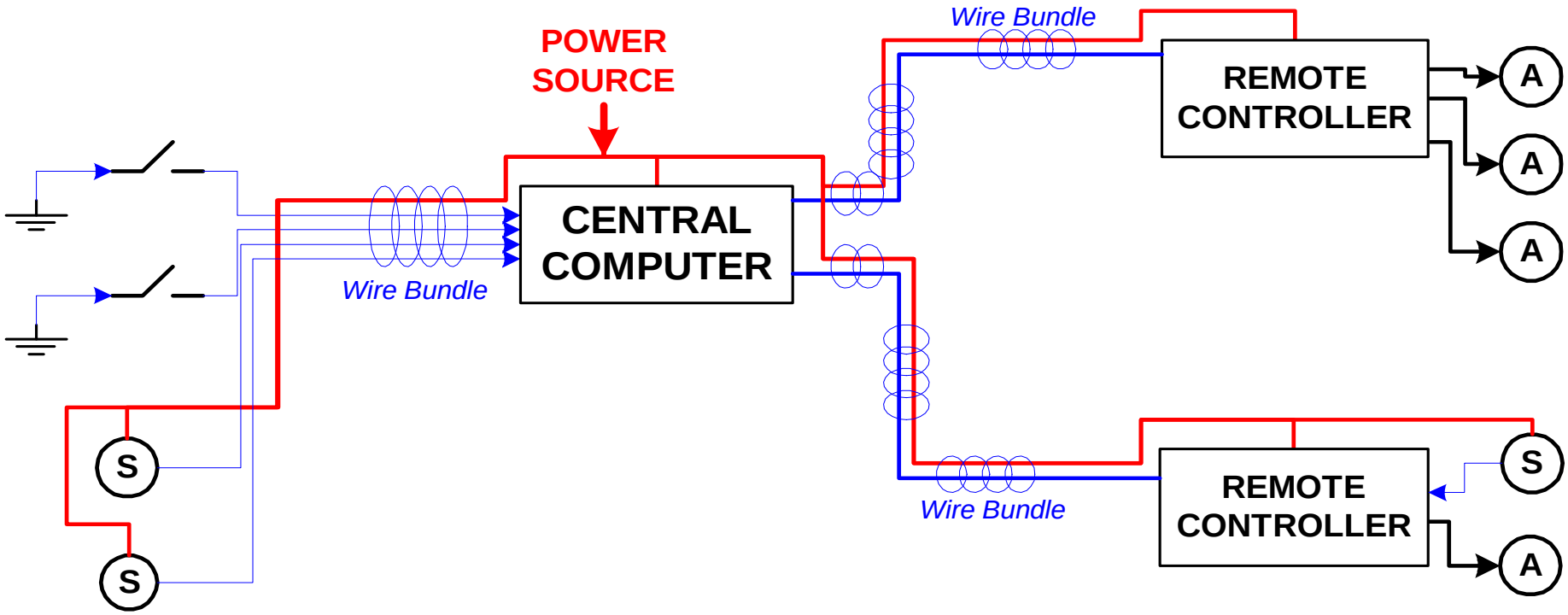
- **Thinner, lighter centralized cable bundles**
  - Pure centralized approach requires all power to touch CPU, causing congestion
  - Control signal wires can be thinner, lighter, and more flexible than power-controlled actuator wires
  - Remote switched power can be run from power supply directly to switching nodes
    - Cable can daisy-chain from region to region
    - Yes, the power cable needs to be thicker since it handles all power loads, but not as thick as a bundle of switched power lines

- **This sets the stage for networking in the next evolutionary step**
  - Remote power switching isn't a convincing win on wiring, but is a good start
  - What if we replace all the analog control wires with a network?

Next, Use Multiple Computers

- **It doesn't take much more to put a CPU at each remote power switch**
  - We already have housing, circuit board, power distribution
  - Compared to that, a $1 microcontroller can be an acceptable additional cost

- **But, analog signals from central CPU to remote CPUs is inefficient**
  - So, let's change multiple analog signals to serial digital communications

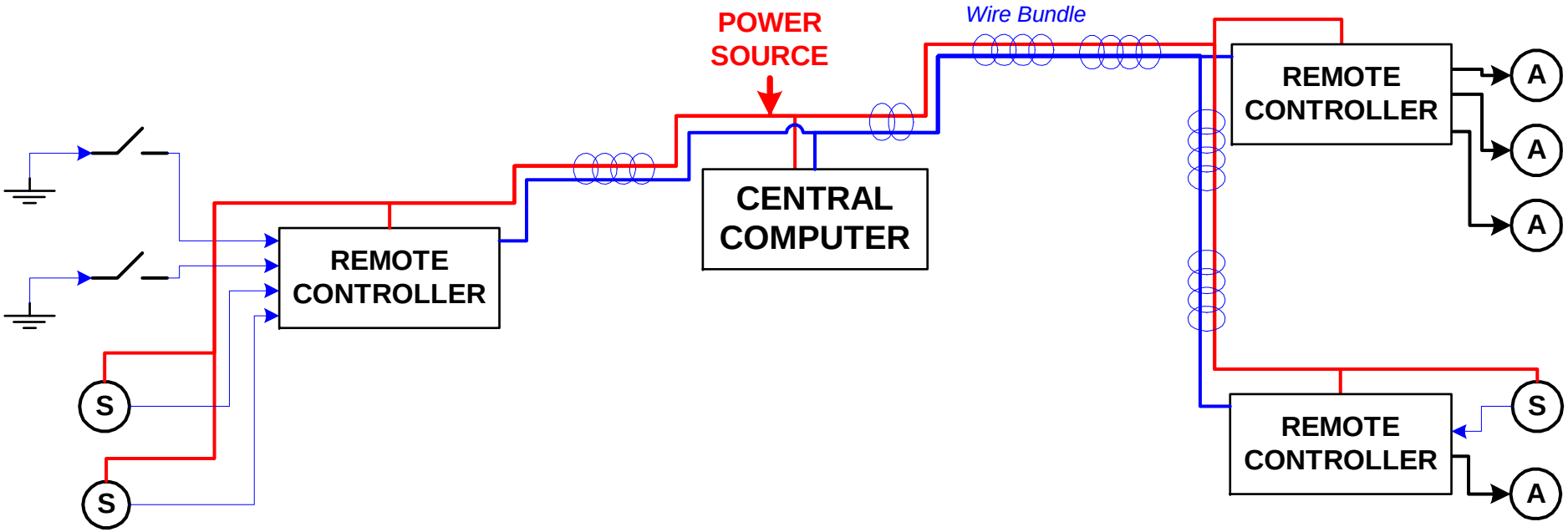


Continue The Progression To Many CPUs

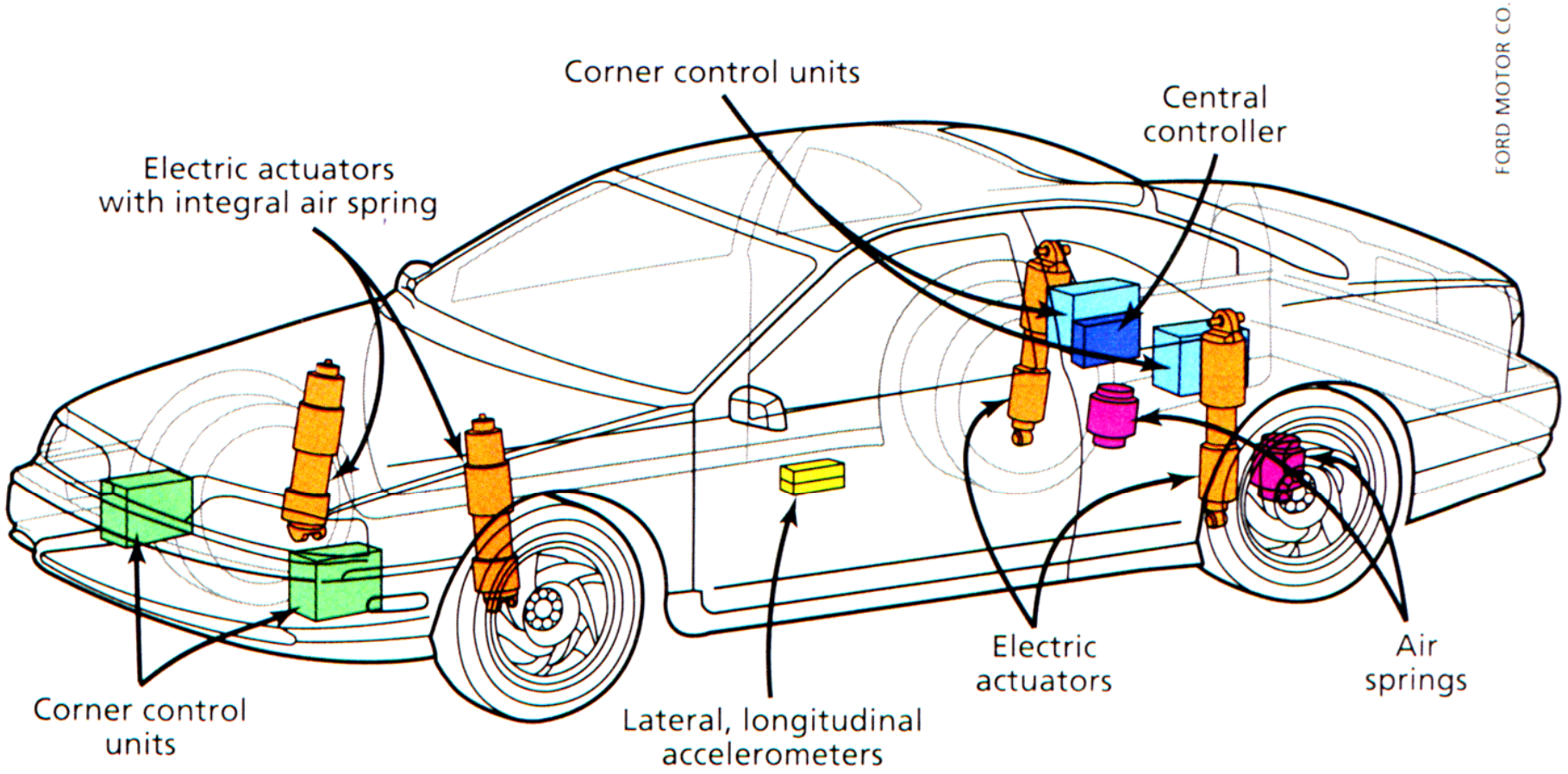
- **Many sensors are near actuators**
  - Sensor/actuator pairs are used to close control loops
  - Many embedded systems aren't that big, so it's hard to be too far from a CPU
- **Add CPUs so that all sensors & actuators connect to a remote CPU**
  - This changes to a fully distributed approach
  - "Central" CPU might not really exist, just "big" and "small" distributed CPUs
  - While we're at it, switch to a network instead of point-to-point wiring

Mostly-Distributed Control Network

- **Both power and control signals flow across a shared wire**
  - Remote power switch nodes have a small computer for networking



Example: Central Control + 4 Corner Controllers

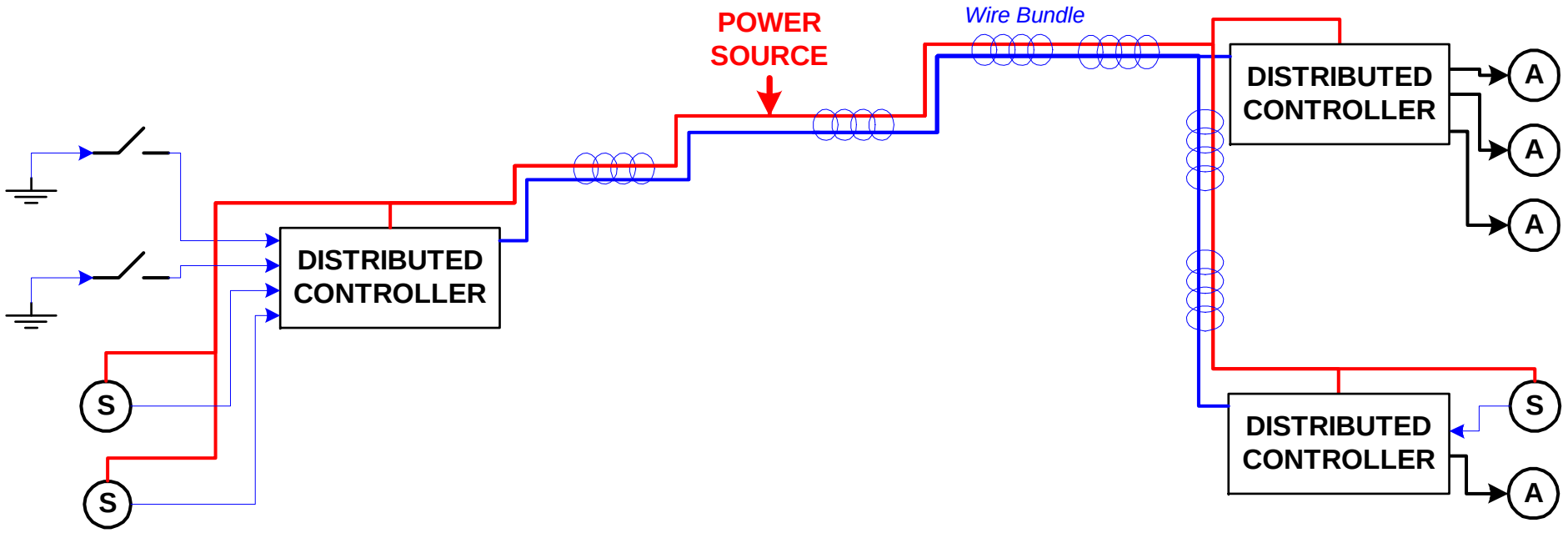


[3] An active suspension keeps passengers on an even keel by sensing accelerations and controlling actuators at several points on the chassis. The suspension system demands high peak power—about 12 kW. Although its average power consumption is a modest 200–400 W, the energy storage components and electrical distribution network must be able to handle the high peak load.

[Kassakian96]

Fully Distributed Control Network

- **Main difference is no central CPU**
  - No “Brain” node calling the shots – all the computation is distributed
  - If all the remote CPUs can handle functionality, do you need a central CPU? (answer depends mostly on the software architecture)



Distributed Embedded Control Tradeoffs

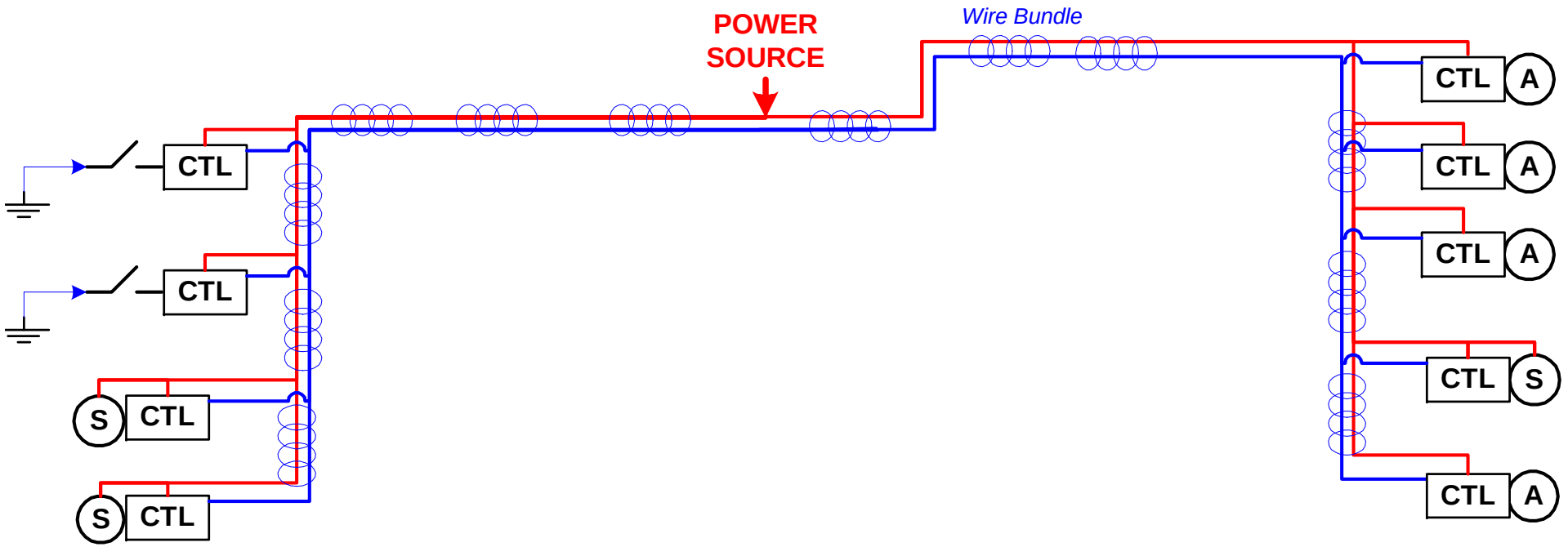
- **Multiplexing control wires saves weight, wire cost, cable thickness**
  - One digital wire replaces multiple analog wires
  - Network must be fast enough to keep control loops closed
    - Much more on this later, but in general this can be done
  - Can use one wire per distribution node if network bandwidth is a concern

- **Network interface computer added to remote switching nodes**
  - Interfacing to even a simple network requires computer-like capability
    - In simplest case, computer just “muxes” wires
  - Local computer’s job is to translate control signals and switch power locally

- **More complicated computers permit functions to migrate**
  - Once we have a remote computers, why not do computation there beyond just network interface?
  - Carried to its logical conclusion, don’t even need the central computer anymore
    - But, doing this requires a significant rethinking of software architecture

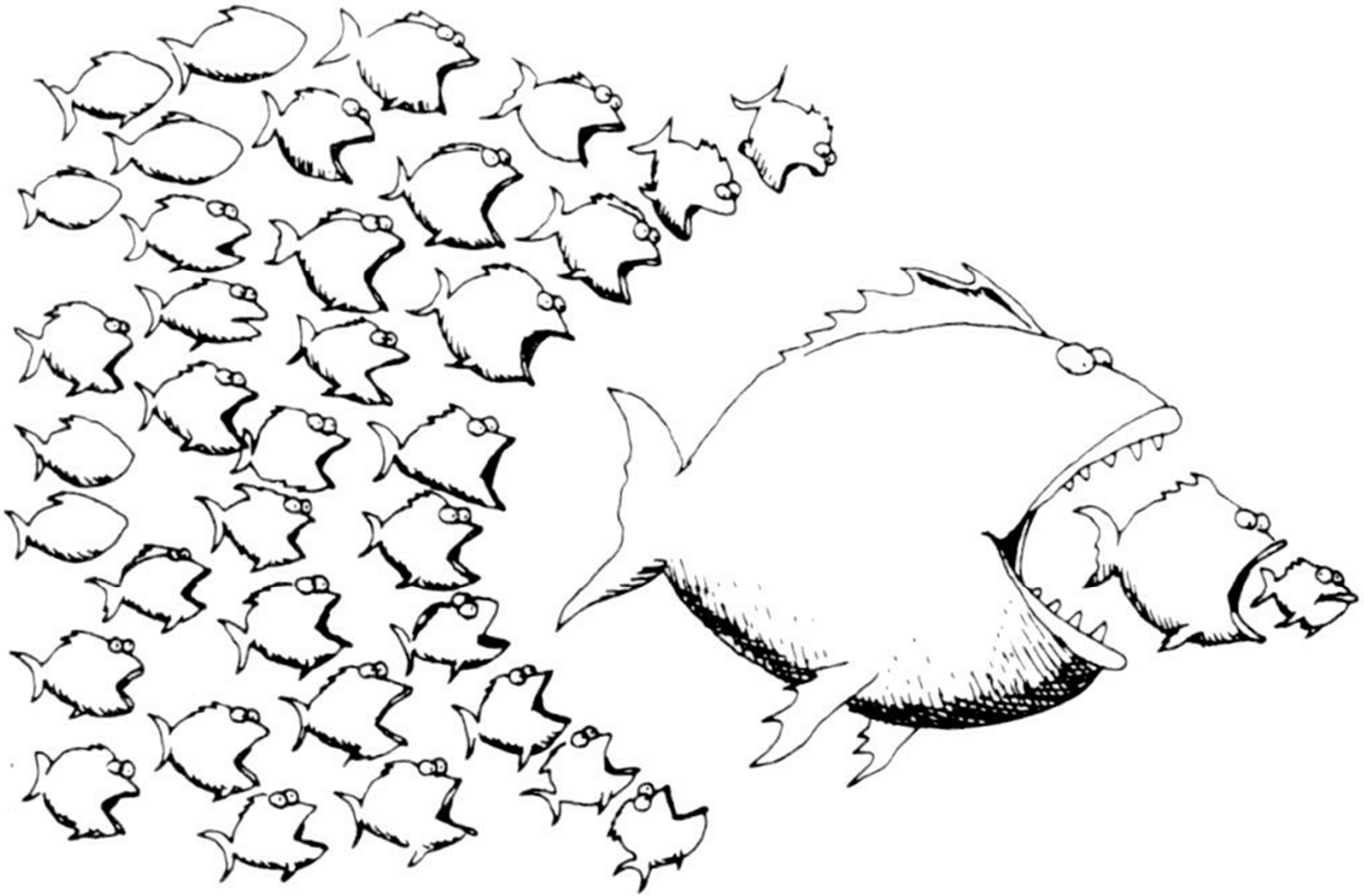
Fine Grain Distributed System

- **Each sensor and actuator also has CPU, power switching, network connection**
  - This is what the course project is all about

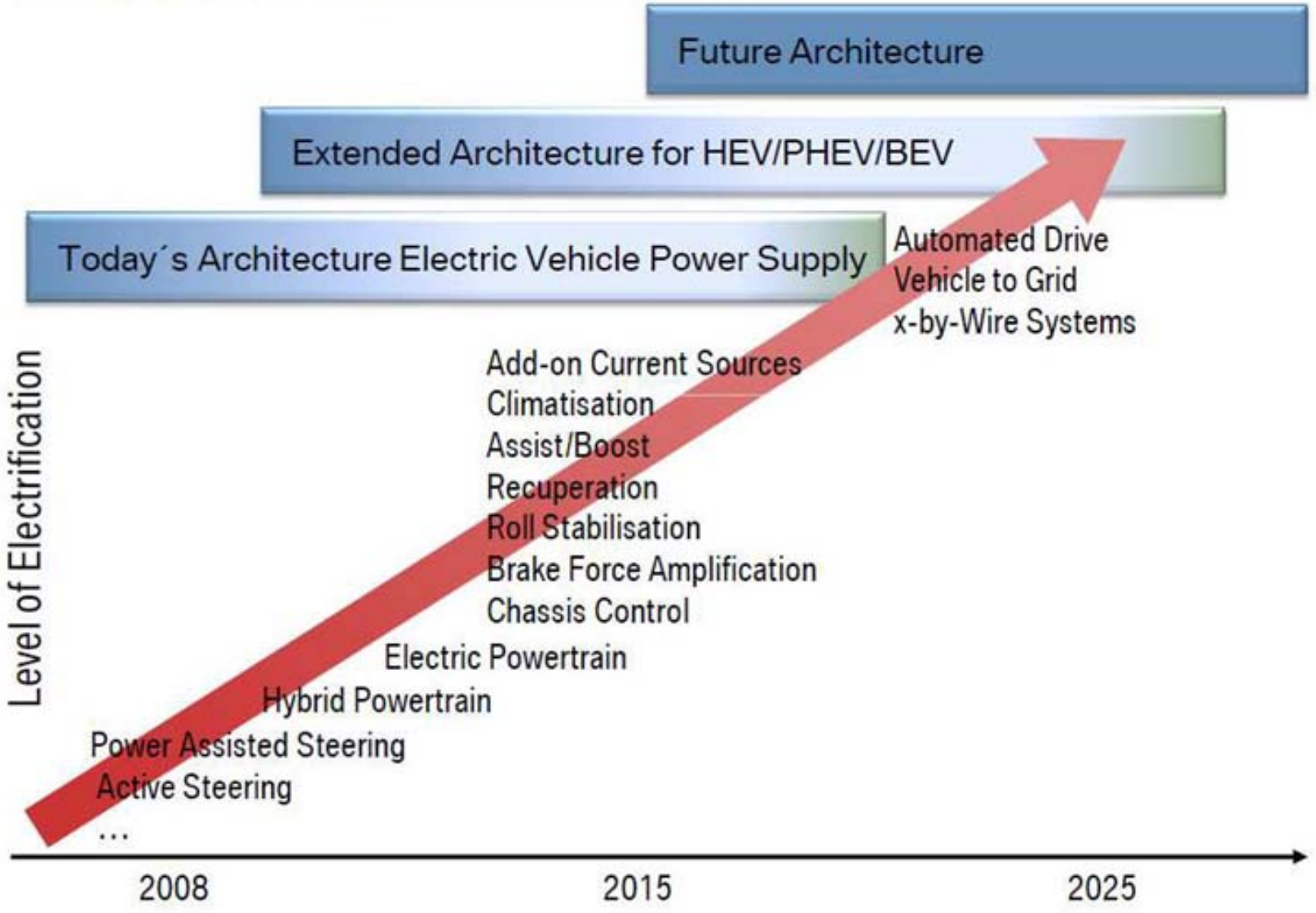


Why Distributed Might Be Better

- **Lots of little things can be better than one big thing**
  - Extensibility / Flexibility / Task Partitioning



INTRODUCTION.
INCREASE OF ELECTRIFICATION.



http://cesa-automotive-electronics.blogspot.com/2012/09/dual-voltage-power-supply-system-with.html

Centralized System Advantages

"Put all your eggs in the one basket and – *watch that basket!*" -- Mark Twain

- **Simple programming model (the one we're taught as undergrads)**
  - Ability to think about distributed architectures is an uncommon skill
- **Powerful CPU(s)**
  - Can use CPU for any needed function (can use desktop PC sometimes)
  - Can adapt CPU loading to operating mode
- **Better operating environment for digital electronics**
  - Put machine in sheltered area away from combustion, environment
- **Arguably simpler software configuration**
  - All changes are made in one place in the system
- **Can grow up to limits of equipment rack**
  - More restrictive than one might think in a harsh environment system
- **Any of these reasons might be sufficient to justify a centralized system**

When Is It "A Wash?" (no advantage)

- **Total system cost/weight**
  - Housing + cooling costs may outweigh wiring savings
  - Distributed system has components in harsher environment than central systems

- **System expandability**
  - Central system has limit on I/O connectors
  - Distributed system has limit on bus fanout (typically 32 nodes)
    - But, arguably easier to install repeaters/bridges
  - Distributed system has limited communication bandwidth (compared to backplane)

- **Inventory costs**
  - Distributed systems have cheaper components, potentially but more kinds of them

Distributed Advantages – Modularity

- **Modular system development, support, and evolution**
  - A different team designing each node
  - Well-defined, tightly enforced interface (system message formats)
  - Can upgrade individual models and limit effect of changes on rest of system

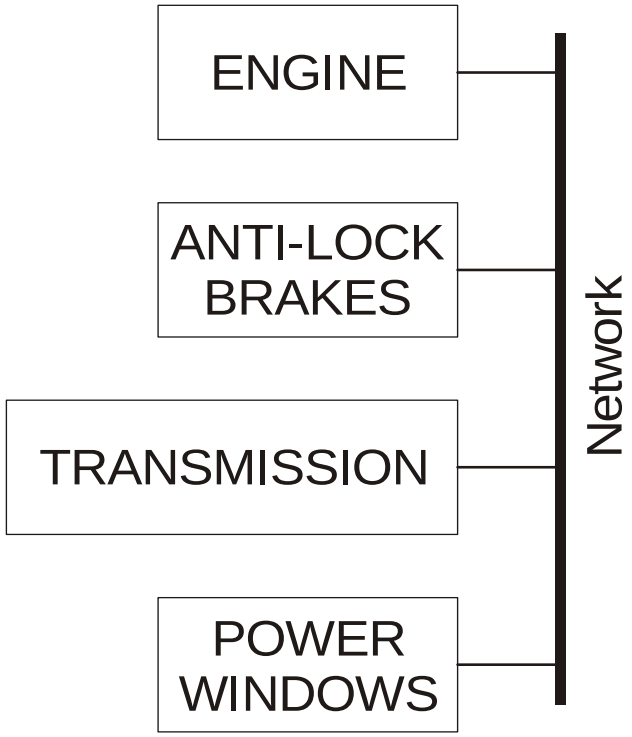
- **Limits competition for resources among different features**
  - Can add compute+I/O power incrementally
  - But, wastes resources on a node that might be inactive most of the time
    - Difficult to “time share” compute resources

- **Reduces interactions**
  - Easier to make worst-case guarantees on a per-module basis
  - Can re-certify only modules that have changed
  - Can have “critical” and “non-critical” modules, reducing certification effort

Example: Car Subsystems

- **Different subsystems often provided by different vendors**
  - Engine, transmission, anti-lock brakes, power windows, etc.

- **Different CPU for each subsystem means:**
  - Each vendor has a CPU all to itself – minimal software integration issues
  - Can change any component without worrying about details of internal software affecting other subsystems
  - Change to convenience subsystem (windows) can be easily shown to have no effect on safety critical subsystem (brakes)



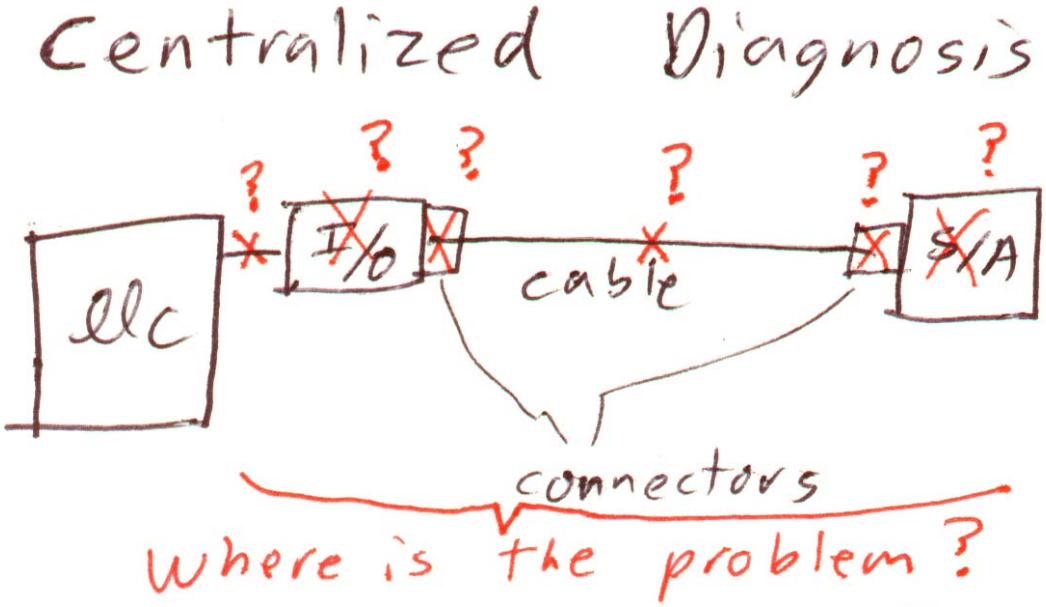
Diagnosability

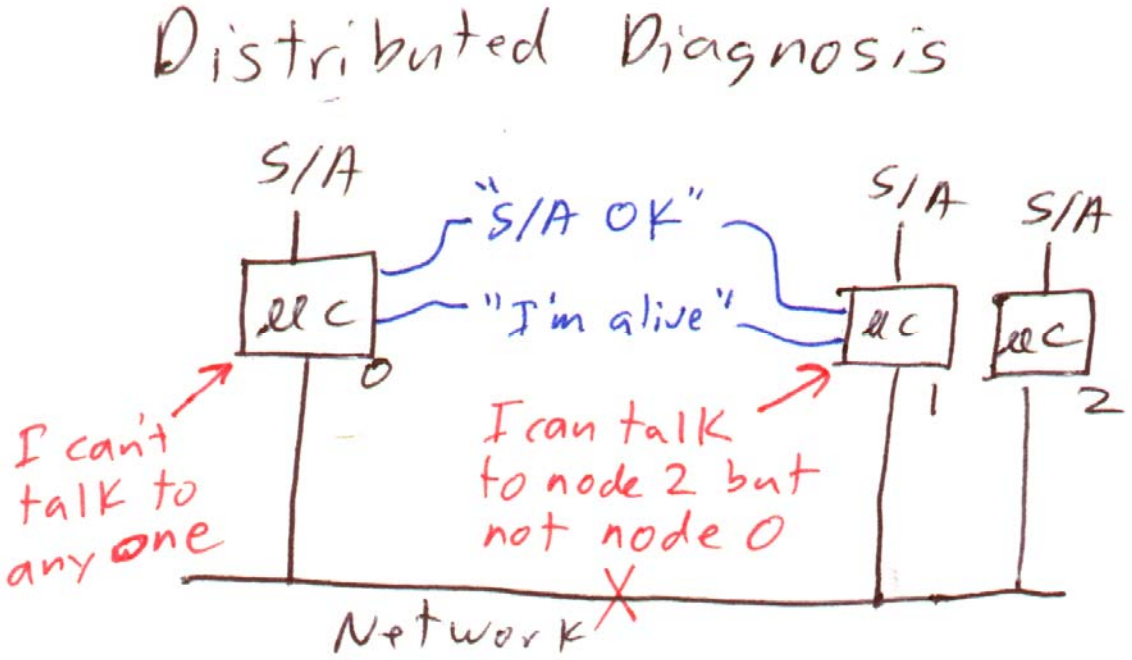
- **Very often this is the decisive advantage**
  - Decreasing maintenance and repair costs can make a big difference!

- **Remote diagnosability**
  - Can isolate problems based on available processors
  - Especially useful on systems where cables and connectors have high failure rates

- **General ideas:**
  - Each controller can diagnose its own sensors/actuators
  - Each controller can diagnose its local connection to the network
  - Each controller can determine which other controls seem to be alive
  - Sampling a few controllers or network segments identifies problem areas

Diagnosability Example





Summary Of Other Distributed Advantages

- **Flexibility**
  - Can modify or upgrade systems by changing components
- **Robust data transmission**
  - Digital network lets you use error coding, controlling noise on signals
- **Simpler to build and maintain**
  - Single bus means you can't hook the wrong wires up – there is only one "wire"!
- **Enables fault tolerance**
  - A single CPU is a single point of failure – multiple CPUs support fault tolerance
- **Improves safety certifiability**
  - Separate CPU for critical functions means non-critical CPU can't cause safety faults

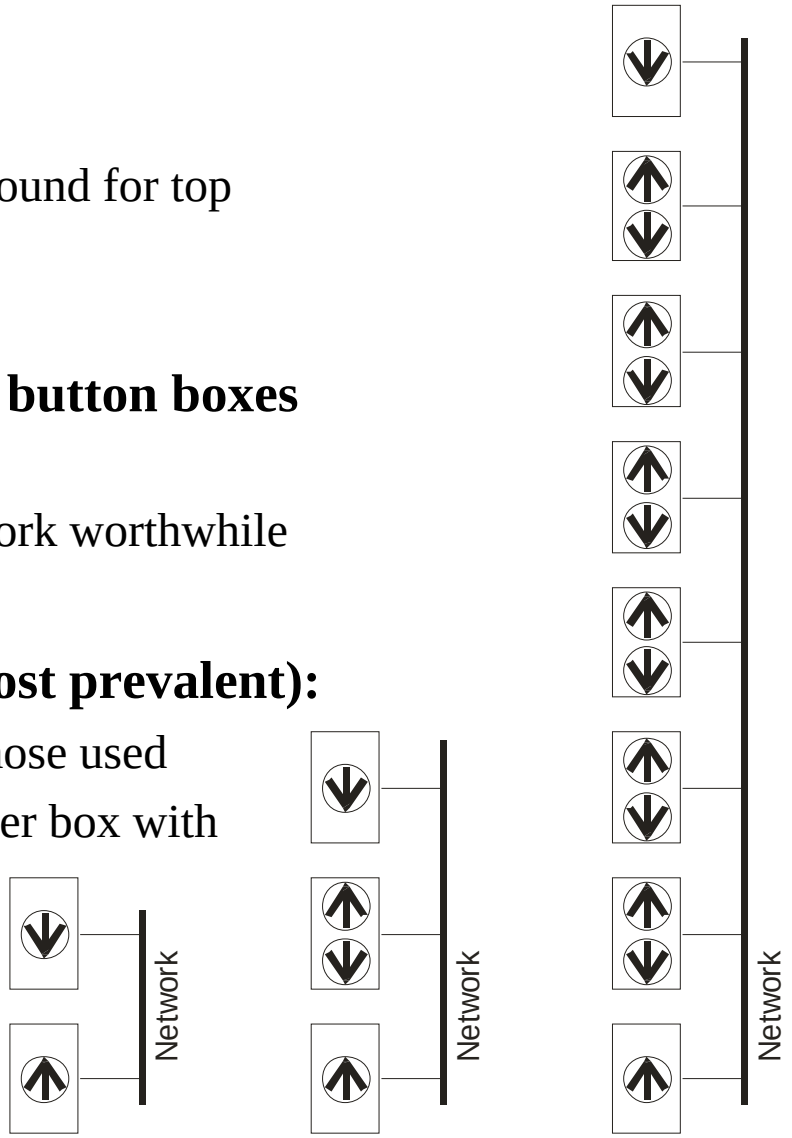
Flexibility

- **Can add new components more easily**
  - Multiple vendors can add components to a well defined HW+SW standard interface
  - New components can have different physical size/shape as long as they can interface to the network

- **Scalable systems can be created on a pay-as-you-scale basis**
  - More copies of components added as system grows
    - (But, there are limits before repeaters are needed for network)
  - But, individual node packaging might be too much overhead if most systems have only 2 or 3 copies of a component
    - A single module with a couple long signal wires might be cheaper than a couple modules with a network wire

Example: Elevator Hall Call Buttons

- **Can build using two standard units:**
  - Two-button unit for middle floors
  - One-button unit for end floors (spin arrow around for top floor)
- **Adding more floors involves adding more button boxes to network**
  - Cost of long wiring runs makes using a network worthwhile
- **But, for 2 or 3 floor lifts (which are the most prevalent):**
  - Middle floor buttons aren't the majority of those used
  - Might be cheaper just to use a single controller box with discrete wires



Elevator Hall Call button assemblies; one button box per floor

(optional details)

Robust Data Transmission

- **Analog data suffers from noise**
- **Digital data can be noise resistant**
  - Error detect & retransmit
  - Error correcting codes
- **Digital sensor/actuator data provides better tradeoffs**
  - Lower bit rate gives better SNR
  - Arbitrarily high precision at cost of bit rate
- **Once you have digital data transmission, you also get a distributed processor**

Simpler To Build And Maintain

- **Single network wire vs. wiring harness**
  - Hard to connect to the wrong wire if there is only one wire
  - Thinner wire, lighter overall weight
  - Far fewer lightning protection devices (if applicable)
- **Maintain by replacing entire node**
  - Potentially easier on-line repair (“hot swap”)
  - Can potentially function with one node broken or missing
- **Potentially takes no space at all**
  - Electronics can be stuffed into nooks and crannies of system
- **Potentially better error containment**
  - If one node fails, entire system does not fail...
  - … as long as network does not fail and node did not have unique data

More Flexible Deployment

- **Buying a centralized computer can be a significant expense**
  - Creates significant barrier for someone on a budget (e.g., a homeowner)
  - Significant investment required before seeing any results at all

- **Sometimes phased deployment/upgrade is better**
  - Limited budget, want incremental improvements
  - Limited down-time for system during upgrades; need phased deployment

- **Distributed systems can help, if designed appropriately**
  - Replace old sensors/actuators with smart ones that are backward compatible
  - Any installed smart systems can provide incremental improvements
  - Can defer expense of central coordinating/optimizing compute nodes until sensors and actuators are in place for them to control

  - In the usual case incremental deployment has higher overall cost
    - But it is often the only practical way to accomplish business goals

Provides Essential Fault Tolerant Capability

- **A single CPU has a single point of failure (the CPU)**
  - Duplicated hardware (multi-channel system) can help improve reliability
  - But a nasty fault or security breech can still slip through
  - And, a 2-of-2 system fails silent, does not fail operational

- **Distributed systems have greater fault tolerance *potential***
  - Different nodes can cross-check each other
  - Breaking into one node does not (necessarily) get you into other nodes
  - If they don't have common mode software failures, system can be more robust

- **Distributed systems can tolerate arbitrary (uncorrelated) faults**
  - Multi-channel architecture without a central voter
  - Can fail operational by consensus voting to exclude faulty nodes from results
  - (See Byzantine Generals problem)

Certifiability

- **Distributing functions potentially encapsulates changes**
  - Changing a non-critical node might not effect critical nodes
  - (But, be careful about indirect changes such as resource consumption)

- **Changing one critical node might not affect other critical nodes**
  - If system components can be certified individually
  - *AND*, each component depends only on advertised interfaces to other components
  - *AND*, change on one component does not change interface
  - Then, *PERHAPS*, this means you don't have to recertify entire system

  - *BUT*, for now, this is a research hope more than a reality
  - It certainly is a way to reduce risk of certification problems by containing changes even if you do have to recertify system just to be sure

Extra-Functional Pitfalls

- **Safety**
  - Making systems safe despite arbitrary failures is difficult
  - (Single-CPU systems generally can't do this at all, however.)
- **Reliability**
  - For a given failure rate per component, adding more components reduces reliability (there are more things to break)
  - This can be compensated for with advantages of diagnosability and redundancy, but be careful
- **Maintenance**
  - Network can help maintenance by exposing key information in a single place
  - Diagnosing concurrent, distributed system usually requires more sophisticated skills, tools, and training

Distributed Tradeoff Pitfalls

- **Distributed advantages are often subtle**
  - Require rethinking of system approach to be a win
  - Can appear as non-functional attributes: (diagnosability, maintainability)
  - May only be beneficial by making new functions easier to add: (flexibility)

- **Distributed systems also have scalability limits, but they are just different than for centralized systems**
  - Electrical fanout of buses / requires repeaters
  - Network bandwidth saturations / requires bridges and careful architecting
  - Complexity of distributed software (different than many are used to designing)
  - A poorly architected distributed system (especially just a porting of a centralized system) probably negates all benefits yet incurs extra cost for being distributed

- **Distributed systems require new skills**
  - Design & debug skills for concurrent, distributed systems
  - Maintenance and operation skills (e.g., network monitoring tools)

Difficult Problems / Research Areas

- **Quantified tradeoffs**
  - It appears that the win for distributed systems is in the squishy areas, not in easily quantified dollars and sense
  - Many systems are driven to distributed at first, then discover benefits once they are there. For example:
    - Automotive driven by need for access to sensors for fuel emission monitoring
    - Automotive driven by physical limits on wire bundle size
    - X-by-wire driven by desire for Byzantine fault tolerance

Why Are Distributed Systems *Different?*

- **Can't just chop up a centralized architecture and distribute it**
  - At least not if you want distributed advantages!

- **Control flow must be decentralized**
  - Close fast control loops locally
  - Attempt to have weak dependence on other units for basic functionality (enables graceful degradation)

- **Data Flow is a limitation**
  - Latency – round trips on a network can cause control lag
  - Bandwidth – inexpensive networks have limited bandwidth
  - Reliability – networks drop packets due to noise, congestion, etc.

- **SW architecture should be compatible with HW architecture**
  - Creating a good distributed architecture is more art than science

Notes On Required Reading

- **Ch 10: Software architecture**
  - We discussed architecture in the last lecture
  - If you are highly distributed you need a software architecture that maps cleanly onto a distributed hardware architecture
  - If you use a "brain" software module and install it in a door lock CPU…
    - … That really isn't a distributed system…
      … It is a centralized software architecture running on a distributed HW platform
      … and at 1 Mbps network bandwidth performance will be terrible

- **Ch 11: Modularity**
  - Modularity Is Good
    - You've (hopefully) heard this many times before, but this chapter reinforces it
  - All the ideas for modules are critical in a distributed system
    - Low coupling: not too many messages between modules
    - High cohesion: all functions involving a sensor/actuator placed near that device
    - High composability: should be able to build many different variants out of a standard set of distributed building block computers, sensors, actuators

Review

- **Distributed architectures bring significant potential benefits**
  - Modularity – system integration of subsystems
  - Flexibility – can scale up system and use heterogeneous components
  - Diagnosability – can isolate faults more effectively
  - Robust data transmission – network enables error control coding
  - Flexible & incremental deployment – no single big box to buy up front
  - Potentially improved fault tolerance & certifiability
  - BUT, common purported advantages often don't materialize (cost, weight, expandability)

- **But, these benefits do not come for free**
  - All aspects of architecture must support distribution (software as well as hardware)
  - Distributed, concurrent design generally requires more sophisticated skills, tools, and infrastructure than centralized designs

- **Sometimes centralized is better**
  - Usually because it is easier to design/implement if you don't care about distributed advantages for a particular application