“Client-Server”; Enforced Modularity

Key Concepts:

1. The limitations of convention-based “soft” modularity
2. The high-level approach of enforced modularity
    a. What are the challenges with implementing enforced modularity?
    b. What benefits do we get from enforced modularity?
3. Micro-services–enforced modularity in the modern world
4. RPCs – the abstraction, the implementation, their benefits, their limitations

Motivation: The limits of “soft” modularity

- Modularity is insanely broad. The boxes and arrows might define:
    - *Soft*: Boxes are functions, clases, arrows are invocations
    - *Enforced/hard*: Boxes are computers, arrows are communication links
- More often than not, we practice soft modularity.
    - It might be ALL that you have practiced up to this point!
    - Why is this a problem???
- Concrete Example: Function call that gets a password:
- Review: how do function calls work?
    - The specifics of this process depend on the architecture’s *calling conventions*:
    - Such conventions are enforced by… compilers, runtimes, etc.
    - Existing systems are variations on a theme: They all save return addresses on the stack and store local variables on stack, though.
    - What can go wrong?
        - What happens if gets receives a huge value?
            - Overwrite other local variables – Data Corruption
            - Overwrite return address – Return Oriented Programming
    - Claim: convention-based modularity is fundamentally brittle and insecure
- **The Crux of the problem: caller and callee share a memory**
    - Question: if we gave them separate naming contexts, could the same problems occur? Depends: can you access an object outside of your context?
    - Yields large propagation of effects:
        - Spurious writes (accidental or malicious)
        - Arbitrary access to modify global variables **(issue with local copies)**
- **The** Crux of the problem: caller and callee share an interpreter:
    - No control over a “runaway module” [prop of effects]:
        - What if get_input executes forever? (infinite loop)
    - No module-level permissions/security [prop of effects]:
        - Validate and get_input fundamentally share a protection level.
    - Cannot manage performance [incommensurate scaling]:
        - The performance of validate is tied to get_input.
    - Modules share a fate in the presence of failures:

        - If get_input fails, so does validate.
        - Impedes specificity in *fault tolerant* (or the idea that the system should survive certain classes of faults)

Could you solve these issues with better languages?

- High-level languages constrain memory sharing (e.g., java, Go, etc.)
- People have tried implementing real systems with these:
    - Biscuit [OSDI 2018] built an operating system entirely in Go
    - Singularity [Microsoft 2000s] built an OS in C#
- Problem: High-Level languages don't seem to solve the interpreter issues
- Problem: High Level languages require expensive runtime support
- Problem: Should you *really* trust these languages?
    - What if they have accidental bugs?
    - What if they are malicious actors?
        - Ken Thompson's Reflections on Trusting Trust
        - Such injections *are* possible! (see https://www.theverge.com/2021/4/30/22410164/linux-kernel-university-of-minnesota-banned-open-source)

Enforced/Hard Modularity

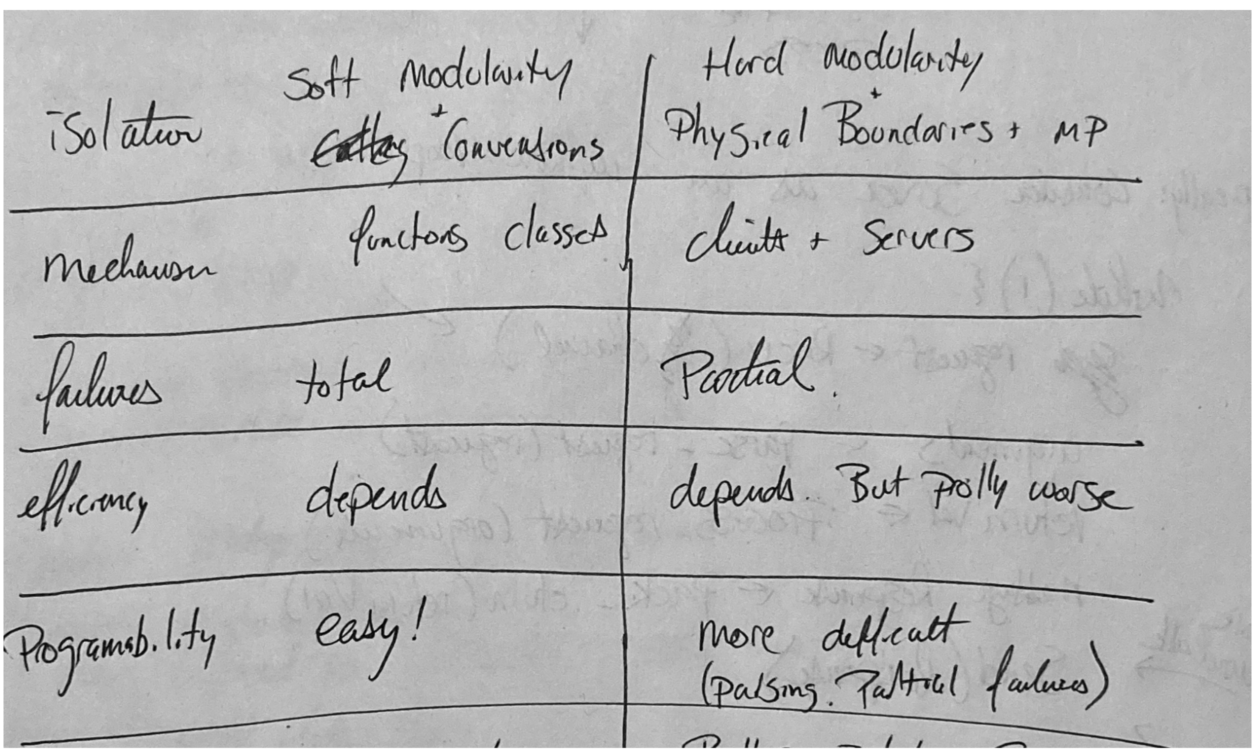
- What if we place them in entirely separate contexts?
- Separate contexts increase *isolation* of modules
    - Independent Memory: No way to modify another modules memory directly
        - Cannot clobber other module's memory
        - Cannot write to other module's instruction reference
    - Independent Interpreters:
        - Contain runaway module (assuming timeouts)
        - Per-interpreter permissions
        - Enable independent scaling
        - Enable less fate sharing and more interesting fault tolerance
    - But, what if they need to talk? Communication links! (Message Passing)
    - Synergistic benefit: Interchangeability. Think of all of the web clients!
- Challenges:
    - Representation: How do computers speak the same language?
        - E.g., little vs. big endian.
        - E.g., "self-contained" messages: messages need to include not just arguments, but also information about specific function, maybe even which client is called (e.g., HTTP put vs. HTTP get)
        - *marshaling* or *serialization*: The processes of converting internal binary data into an externally consumable format
    - You have to parse each message! No compiler :(
    - You *get* to handle failures. But also, you *must* consider them!

    - Comparatively difficult to change a protocol
- Benefits:
    - Scalability
    - Better fault tolerance-performance tradeoffs
    - Trust

RPC: reducing the developer burden of using Client-Server

- Abstraction–execute some code on another computer
- Implementation:
    - REST:
    - Protobuf, Apache Thrift, etc.
- This seems great! Abstraction to the rescue, right!
    - Developers never have to serialize again!
    - All the peculiarities of message-passing are hidden from the user!
- Problem: is RPC the same semantics as functions? Why or why not?
    - Performance! Serialization is… Expensive. Unpredictable.
    - Semantics! Functions give you *exactly once semantics*. Can RPC?

Summary Table



isolation	Soft modularity + conventions	Hard modularity + Physical Boundaries + MP
mechanism	functions classes	clients + servers
failures	total	Partial.
efficiency	depends	depends. But prolly worse
Programability	easy!	more difficult (parsing. Partial failures)