1. Foundations

This chapter presents the intellectual foundations for Joule: first, a brief synopsis of the history of programming languages, with emphasis on characteristics relative to distributed systems and Joule; second, a checklist of the criteria for distributed object programming languages (criteria which motivated the development of Joule); and third, a recap of some familiar, well-established design principles, showing how Joule embodies these principles in its design and supports the application of these principles to programs written in Joule.

1.1. Antecedents

1.1.1. The Rise of Modularity

From straight-line code to procedures to objects, the history of programming languages has been a history of increasing modularity to help solve increasingly complex problems. Modularity makes interfaces between pieces explicit, so that the extent to which the separate pieces interact can be controlled, then minimizes the dependencies required for a given level of cooperation. The more extreme the modularity, the more the unintended dependencies between the parts can be avoided. As systems get more complex, these interactions start to compound, placing an upper bound of complexity on the sophistication of programs and the size of a problem they can solve.

With procedures, programmers created boundaries around packages of behavior, allowing them to define procedures once and then not worry about the implementation when using those procedures. Factors such as data interactions in global environments still led to unintended interactions and a limit on the sophistication of programs.

With abstract data-types, programmers created boundaries around static packages of data and behavior, increasing the sophistication in each "black box". Programs could now manipulate entities representing abstractions relevant to the problem being solved.

With objects, programmers created boundaries around dynamic packages of behavior and state. The polymorphism of object-oriented programming enables a much stronger separation between interfaces and implementations, allowing black boxes to hide not just the details of implementation, but also the details of which of many implementations the black box represents. The complexity limitations come from

the vestiges of global environments and the difficulty of synchronizing on shared data in a concurrent environment.

In the book *The Mythical Man-Month* **[14]**, Frederick Brooks described a study in which it was discovered that programmers wrote the same number of debugged lines of code per day no matter what programming language they were using. This observation motivated the drive towards higher-level programming languages that culminated in languages like APL and PL/1: the more and higher-level the abstractions they could make accessible, the more effective each line of code written by the programmers would be. At the same time, a much smaller thread was pursued in, for example, the Lisp community, leading through Simula, Smalltalk, and C++: the thread of abstraction languages. Instead of building in particular high-level abstractions, this new class of languages provided tools for programmers to build abstractions extending the language itself. In these languages, each line of code contributes to the level of abstraction of the next line of code. As a result, even though they start with much less capability, object-oriented systems have accumulated enough leverage that they are more effective tools for programming large systems than the high-level languages: programmers write the same number of lines of code, but they write more expressively.

By explicitly recognizing these threads of increasing modularity and abstraction power, Joule takes the next steps along both dimensions. Many of the capabilities of Joule were discovered by examining existing large systems such as networks and operating systems for the modularity tools and abstraction mechanisms which give them organization, and so make them manageable. The techniques that worked have survived into many existing systems; the techniques that failed have either fallen by the wayside or been entrenched in existing systems and demonstrate obvious failure modes.

1.1.2. Distributed Object Programming

Distributed object programming brings powerful new capabilities to computing. However, these capabilities demand the unlearning of some important paradigms of previous programming languages. The most important change required in the programmer's thinking is the abandonment of sequential call/return control flow. The sequential control flow and call/return is very natural for procedural divide-and-conquer programming in which each procedure calls other procedures in order to accomplish a specific task. The stacking behavior inherent in call/return is less appropriate for object-oriented systems in which the objects have invariants that need to be re-established before another call can be made to the object. In Smalltalk, for example, if a loop run on a collection of objects removes (or causes to be removed) an object from the collection, most times the loop operation will fail because it didn't expect the arrangement in the collection to change during the iteration.

The problems of sequential control flow and stack-style call/return are even worse in distributed object systems because such systems inherently provide concurrency and asynchrony. Sequential programming languages fundamentally cannot support distributed object systems; sequential programming languages plus external operating system support can, but the difficulty of developing applications, and the con-

tinued delicacy of communications between machines, suggest that existing tools are not well-suited to distributed object systems.

An analogy about stacks that is appropriate to objects is that stack-based programming is like a person whose work patterns are interrupt-driven. The introduction of a new task forces the current task onto the back burner. Interrupted tasks accumulate in the back of such a person's mind like calls on a stack. The mental model associated with a stack forces all operations into a LIFO queue regardless of the logical relationships between the tasks or their importance. A trivial but time-consuming task may be performed before a more important one simply because it was initiated later.

Distributed object programming is much more like managing one's time with a "to do" list. New tasks can be added to the task list without interrupting the execution of current tasks, and tasks can be interleaved without interfering with each other. Dataflow synchronization is like inter-task dependencies. It drives the ordering of tasks on the to-do list. "I have to cash my paycheck at the bank before I can buy this week's groceries, and I have to buy detergent at the grocery store before I can do the laundry." Tasks that are not logically dependent do not interfere with each other: "I can cook dinner regardless of whether I have done the laundry."

This property of distributed object programming derives from the defining characteristics of asynchrony, concurrency, and fundamental support for communications. Distributed systems are inherently asynchronous because they have to deal with events arising from multiple sources at spatially separated sites. The architecture of a distributed object programming system must be able to cope with this asynchrony (so that, for example, multiple clients can make requests of a service simultaneously). Distributed systems require concurrency because they operate on many machines simultaneously. Finally, because distributed systems must allow and encourage interaction between sites, they need to support communications abstractions. Although such support can be implemented between separate sequential "threads" at multiple locations, the sequential model adds nothing to the ease of implementing communications abstractions and distributed systems.

1.1.3. Server-Oriented Programming

Distributed object systems are rare today because building robustness on top of today's network software is difficult. *Server-oriented programming* (SOP) realizes the advantages of distributed object programming, by making the environment sufficiently resilient for distributed objects to survive. SOP applies intuitions about client/server systems to all levels of programming.

Take, as an example, a database system on a network. Multiple clients access the database across the network. These clients run concurrently with the database; they send requests, and occasionally wait for the answers, but otherwise remain responsive to the user. Such a client might access multiple databases on more than one machine.

Now, apply these same intuitions about the relationship between the clients and the database server, but within the database: there's a request-handling server for each client user, a disk subsystem, and an

indexing engine. The request-handling servers may perform translations on user queries before calling on the indexing engine with them. Within the indexing engine is a query optimizer, a B-Tree manager, and a transaction handler. Within the disk server is a process for each physical disk drive, a replication manager to protect against media failure, a transaction log, and a page reshuffler for grouping pages that should be clustered. Within each disk-drive process is a disk-arm scheduler, a disk cache manager, and a device controller.

Each of these units communicates, via well-defined protocols, with the other modules—and within each unit, sub-units communicate using other well-defined protocols. At each level of granularity, from the network application to the database to the disk handler to the device handler, the same client/server intuitions apply: separate servers can run concurrently and schedule and handle requests from other servers. Each server is a black box so far as its clients are concerned—the request handlers need not know which internal servers make up the indexing engine, so long as it responds appropriately to requests. The particulars of its internal structure are irrelevant, and may even change over time.

Proper encapsulation is a requirement for the creation of robust servers. A *robust server* is one which can guarantee continued correct service to well-behaved clients despite aberrant (that is, arbitrary or malicious) behavior from other clients. This is in contrast to fault tolerance, in which servers are able to operate continuously despite component failures (failure of other servers, or of hardware components). The kernels of traditional operating systems are designed to be robust—when one application misbehaves or crashes, the operating system is supposed to continue uninterrupted service to other applications.

The concept of robust servers is a very powerful tool with which to distinguish application platforms. They can't be built in most systems.

Attempting to implement distributed object programming over today's networks reveals problems that already exist, hidden, in single-machine systems. For example, a single misbehaving application can degrade the performance of other applications by disrupting services on which they both rely (causing "thrashing" of virtual memory, or allocating too much disk space). In a robust system, applications would be able to cope with temporary unavailability of those services and perform productively while waiting for them to return, rather than seizing up. Server-oriented programming builds tools to deal with the problems rather than just hide them.

Many of the principles of Joule were distilled from existing systems, and could be applied in more than just a language context, improving the robustness of more traditional applications.

The foundations of server-oriented programming enable extremely long-lived systems. These systems must meanwhile be able to grow and change, which motivates another defining characteristic of server-oriented programming, *open entry*—the capability of adding new components or replacing old ones, in a running system, with no interruption of service. This both requires and enables full encapsulation—a new server can replace an old one, despite having a completely different internal structure, if and only if its protocol is upward-compatible with that of its predecessor. To the clients, no change is visible.

These properties apply at all levels of a server-oriented system, enabling reliable construction of large and complex systems by assembly of well-behaved components. Properly robust servers in such systems could independently recover from failure, and communicate with each other through a well-defined interface such that they have no interactions beyond those that are explicit. The restriction of inter-

server interaction to explicit exchanges within a well-defined protocol forms the basis for real security in server-oriented systems. Security is discussed in more detail in Chapter 8.

1.1.4. Market-Oriented Programming

While server-oriented programming allows programs to guarantee the correctness and availability of computing services provided to clients, market-oriented programming enables systems to be adaptive to user and client needs and available resources by introducing *agoric resource management*. Agoric resource management uses market principles to dynamically allocate resources among software agents. By introducing the equivalent of money into the software resource management process, Joule takes advantage of the institutions and abstractions that have been developed for managing the allocation of physical goods.

For resource management issues to which markets are not well-suited, traditional centralized control (as demonstrated in the late USSR, for example) can also be constructed from the resource ownership and transfer foundations.

Markets work in the physical world because, in a sense, they already form a distributed computing system. Agents exchange goods for money, and in the process produce information about how valuable those goods are, in the form of prices. The role of money in a market system is as an abstraction which represents access to resources. Agents in a market make their decisions based on local knowledge of prices and the availability of resources. The information resulting from those decisions—what to buy and at what price—propagates through the market (the retail prices a consumer is willing to pay affect the prices which retailers are willing to pay wholesalers, which in turn affect deals between wholesalers and manufacturers). The communication of these price signals enables the whole system—the market and its participants—to allocate resources in a way that adapts to changing conditions and the different needs of different agents more effectively than could be done by any single allocating agent, based on more information than any such agent could access. (The costs of gathering and processing such information centrally would be prohibitive; much of the information would be out of date before it reached the central allocator; and in the context of mutually untrusting programs, such a central allocator might not be trusted by the participants.)

The introduction of market principles to server-oriented programming systems provides a necessary framework for efficient, decentralized management of computational resources. Local (in time or space) shortages of resources represent an opportunity for load-balancing agents—arbitrageurs—to correct the imbalance at a profit. Such agents need not violate the modularity of the programs they're helping—this resource allocation can be done through voluntary trade using client/server communications protocols, as will be seen in the next section.

Market-oriented programming relies on two concepts: the encapsulation of resources—that is, ownership by particular processes of access to blocks of, for example, memory or processor time—and the communication of access to those resources, making such ownership transferable in a flexible manner. This enables a simple initial allocation of resources among a set of providers (as described in Chapter 9) to evolve in complexity in response to the specific demands made on the system.

Encapsulation and communication of resources enable performance to be guaranteed by allowing programs to reliably purchase the rights to

particular quantities of resources at a future time. This enables servers to commit to deadlines for providing computational results to clients. Encapsulation of access (ownership) lets programs control resources; communication of access lets programmers build facilities that dynamically allocate those resources.

Object-oriented programming separated *what is to be done* (communicated by inter-object messages) from *how it is to be done* (the methods, invisible from outside the object, that are enacted in response to each message), so that the calling object need have no knowledge of the internals of the called object. A closely analogous benefit arises from the separation of resources from prices. The use of a medium of exchange (money) enables ready conversion between different kinds of resources—memory and processing cycles, for example—which would otherwise be incomparable.

The need for a particular resource varies with time and with the function being performed—suppose a particular 3D graphics rendering package is CPU-intensive. While it is running, the price of processor cycles goes up relative to memory, so other programs can adjust their budgets to rely more heavily on memory than on CPU by, for example, using more caches. If, instead, the system is currently dominated by a memory-intensive process like a drawing program, physical memory becomes expensive, making virtual memory more attractive. Resource management for complex systems needs to provide this same ability to allocate multiple kinds of resources among multiple users with diverse needs who contend for those resources.

1.2. Rules of the Game

In the context of market-oriented programming, we require a simple set of rules so that servers can interact with each other predictably. This is best illustrated by the observation that *a business can be open to the public because its cash register isn't.* Supporting such businesses requires strict, understandable rules so that participants can successfully protect their own interests while cooperating with other parties.

A computational foundation for supporting the interaction of diverse parties also defines the "rules of the game" by which those parties can interact. One never finishes learning the patterns which emerge from the rules of an interesting game, but it is important that the rules be simple enough to be understood completely, particularly if real interests are at stake.

The relevant systems are the frameworks for interaction. The C language, for example, does not support an open system because programs written in the same C address space can corrupt each other. C plus UNIX gives better support because it provides processes some measure of protection from each other. However, the continual security problems on the Internet (exemplified by the prevalence of "firewalls" that deliberately cripple insecure communications) demonstrate that C plus UNIX still does not support open systems because it is too insecure.

We define an open system as one which can continue to operate while allowing untrusted parties to "join the game," as opposed to the sense

of "open system" in which any server can get inside any other server, including its cash box.

The design process for Joule was based on finding a minimal set of rules that all processes could count on. Everything else necessary for large scale programming could be built in the framework of the fundamental rules. The computational model presented in Chapter 3 describes the rules for everything except resource management.

Joule is certainly not unique in this regard. It is unique in the set of constraints that were applied to guide the design process to a set of rules.

The checklist for Joule combines the checklists in **[65]** and **[89]**, the principles presented in Section 1.3 below, and practical issues from building large systems. Here is a partial informal checklist that drove particular aspects of the design of Joule:

- *Encapsulation and communication of information, access, and resources*

  Without this safety, businesses can't open their doors, users can't manage their resources, and groups can't cooperate.

- *Principles scale to arbitrarily large systems*

  There should be no inherent bottlenecks such as global state or inherent distributed transactions.

- *No global knowledge, control, or trust*

  These would all prevent the cooperation of agents that don't trust each other, and they are all single points of failure.

- *Robust servers*

  i.e., servers that can guarantee continued correct service to well-behaved clients despite aberrant (that is, arbitrary or malicious) behavior from other clients

- *Open entry*

  New services can be started, new customers can connect, and so forth.

- *Security*

  i.e., trust management so that services can interact while maintaining encapsulation boundaries

- *Composable correctness*

  It's possible to build something that fulfills its contract relying only on the contracts of other servers.

- *Separate resource management*

  This is the familiar principle of separation of concerns, but applied to a concern that most systems give users very little control over. This is the foundation out of which agoric resource management can be built.

- *Efficient execution*

  The model must be expressive, but must also map well to existing computer hardware architectures (for example, using message sending in Joule to implement procedures must be as fast as traditional procedure invocation).

- *Self-basis*

  It should be possible to build the distributed system in itself. If not, then the system doesn't provide sufficient functionality for managing distributed systems. Further, no single policy for how to distribute programs can be right for every application, so it

must be possible to express different distribution solutions in the language.

- *Concurrent and asynchronous*

  This is an inherent property of networks of machines that should be supported directly in the programming model.

1.3. Elements of Joule Style

This section restates some familiar, well-established design principles to show how Joule embodies these principles in its design and how it supports their application to programs written in Joule.

1.3.1. Recursive Abstractions

As described previously, the use of the client/server orientation at all levels of program design gives Joule many of the required characteristics for creating robust systems. This can be generalized to the principle of *recursive abstractions*—the use of similar organizing principles at different levels of operation. This property allows efficient scaling of code—the techniques learned for building small programs work equally well for large programs.

Another example is the principle of transparency—at all levels, Joule clients don't need to know the true nature of the server with which they're dealing; it may be a composite server or merely a transparent forwarder that chooses among several competing servers. This application of the same abstraction at different levels of granularity makes for more powerful and well-behaved Joule programs.

1.3.2. "What"/"How" Separation

This is a familiar application of the principle of separation of concerns— separating the interface of a service from its implementation to achieve better modularity. The interface specifies *what* is to be done—for example, what services a client requests from a server. The implementation—the *how*—provides the service, but the particulars of the implementation are not determined by what was requested; the internals of the server can be any implementation that conforms to the communication protocol and reveals correct results. An example of this was described above, in the discussion of separation of messages from methods in object-oriented programming. This is another organizing principle for programs which is well-suited to the capabilities of Joule.

Encapsulation is the property of Joule that hides the "how"—the limitation of interaction between processes to explicit exchanges prevents the calling process from discovering details of the implementation with which it is interacting. Polymorphism makes the "what" (the message, or the service requested) independent of a particular "how"—the server can choose internally among multiple techniques for doing the work itself, or even subcontract for the service elsewhere, with no difference apparent to the client.

1.3.3. Mechanism/Policy Separation

The *mechanism* of a particular function—the features present at the lowest level of abstraction to enable that function—should not inherently

impose unnecessary limitations on the range or application of that function. When there isn't a single "right" answer, Joule provides frameworks in which many policies can coexist. The restriction of the uses of a function—the *policies* governing its use—should instead be reserved for explicit definition at higher levels of abstraction. Caching strategies are a good example: the most effective strategy varies with how the cache is used.

The usefulness of this separation comes from abstracting from a set of desired capabilities the kernel capability which is most fundamental. An example of this is time-slicing. The fundamental capability is "determining who has control of the processor when." A system that dictates time-slicing at the kernel level is overdetermined; it rules out real-time applications, for example (as commonly defined). Joule instead treats ownership of the processor as a fundamental abstraction, allowing time to be sliced if and as needed, but also allowing for real-time applications to be built in Joule. This generality allows experimentation with other abstractions besides time-slicing, such as deadline scheduling.

Mechanism/policy separation is a way of separating things to create a new domain for distinctions. Putting the most general abilities at the bottom of a hierarchy of abstractions creates layers which can be used to determine the abilities of the layers built on them, as needed, rather than being inflexibly locked in from the very lowest layers on up.

1.3.4. Composable Orthogonality

The design of the Joule kernel is intended to separate functions along natural lines that allow the resulting abilities to be distinct, and to provide synergy when combined. The criterion for separation of functions is orthogonality: no function should partially duplicate the capability of another. This is akin to orthogonality in a mathematical coordinate system: from an orthogonal set of basis vectors, any vector in the space can be constructed more simply than from a non-orthogonal basis.

In Joule, this clean separation of powers results in smaller abstractions that give more power. Two structures that overlap in their abilities often conflict when used together in some ways. The lack of overlap between facilities in Joule prevents the elements of the Joule kernel from getting in each other's way—they can be sensibly combined in any way without conflicting. Also, if two structures overlap, it reduces the space of abilities that can be accessed by combining them—because they partially reproduce each other's abilities, less new function is discovered by using them together. The combination of two orthogonal functions, however, creates a space of new abilities inaccessible with just one of the components.

This partitioning of function is a design criterion at every level of Joule. For example, the **ForAll** statement (introduced in Section 5.3) implements multiple instantiation of code; the `choose:` message implements conditionals. There is no way to implement **ForAll** using `choose:`, or to implement `choose:` using **ForAll**, yet the two, combined, generate much of the Joule language. At a higher level of abstraction, resource management and concurrency are orthogonal facilities—neither can be used to generate the function of the other, but combined they give a whole new set of powerful abilities.

1.3.5. Complete Virtualizability

Anything virtualizable must be completely so, but not everything must be virtualizable. Numbers can be completely virtualizable without Tuples being so. However, in Joule, *everything* is completely virtualizable.

Complete virtualizability is one of the primary mechanisms for composable orthogonality. Because clients can only interact with servers by passing messages, other servers (middlemen) can be interposed between a client and a server to add functionality without requiring a change to either the client or the server. Virtualization only aids composability if it is complete: if programs behave differently in the presence of middlemen that weren't *intended* to change the behavior, then middlemen cannot be transparently added. Complete virtualizability enables *transparent* layering of functionality: servers only affect each other in intended ways.

One of the driving examples for transparent layering is a distributed version of Joule built in Joule. Each message from a client goes to a proxy for the intended server. The proxy turns the message into data which it sends to a handler on a remote machine. The handler on the remote machine turns that data back into an equivalent message and sends the message to the actual server with which the client wanted to communicate. Complete virtualizability means that neither the client nor the server can observe whether the network forwarder is there—if they could, then the client or the server could be written in such a way that it breaks in a distributed system but not on a single machine. The ability to observe the forwarder would require every program to take into account the implementation of the distributed system. With transparent layering, every Joule program can run across a network with *no change*.

Complete virtualizability is a very stringent requirement for the language: operations on numbers must succeed even if the "numbers" are forwarders to numbers on other machines, or user-defined servers for complex numbers, arbitrary-precision real numbers, or fractions. Message sending must work even if the "messages" are user-defined servers that merely act like messages but are actually implemented some other way. The techniques with which Joule satisfies these requirements while remaining efficient to implement have been designed. Some of these mechanisms will be presented in this document. A simple example is the primitive addition operation for Integers: if it is supplied with a non-Integer addend, it sends the `+from-Integer` message to the addend, supplying the original receiver (now known to be a primitive Integer) as an argument, along with the original result channel. The original addend (a complex number for instance) can then supply the behavior for adding itself to an Integer. This is not the complete story for bottoming out operations on numbers, but it demonstrates one of the simple techniques.

The completeness of virtualizability in Joule allows programmers to transparently extend functionality anywhere. They can build new transparent layers (such as the distributed system), or they can extend the functionality of any of the system abstractions (numbers, channels, messages, and so forth), while preserving the transparent layering properties of the system. Virtualizability is implemented largely through anonymity and polymorphism: servers can be distinguished only by their actions in response to messages. Security sometimes requires certification, however—you want to deposit only in your bank account, not some forwarder that might redirect your money. Joule

both supplies certification, which must be used carefully to preserve virtualizability of abstractions built with it, and provides abstraction to support virtuality in the presence of certification.