Tinkertoy: Build your own operating systems for IoT devices

Bingyao Wang
University of British Columbia
Vancouver, Canada
bingyao@cs.ubc.ca

Margo Seltzer
University of British Columbia
Vancouver, Canada
mseltzer@cs.ubc.ca

Abstract**—The Internet of Things (IoT) makes it possible for tiny devices with sensing and communication capabilities to be interconnected and interact with the cyber physical world. However, these tiny devices have limited computing power and memory, so they often cannot run commodity operating systems, such as Windows® and Linux. IoT devices are deployed everywhere, from smart home appliances to self-driving vehicles, and their applications impose ever-increasing and more heterogeneous demands on software architecture. There are many special-purpose and embedded operating systems built to satisfy these wildly different requirements, from early sensor network operating systems, such as TinyOS and Contiki, to more modern robot and real-time control systems, such as FreeRTOS and Zephyr. However, the rapid evolution and heterogeneity of IoT applications calls for a different solution. Specifically, this work introduces Tinkertoy, a collection of standard operating system modules from which developers can easily assemble customized operating systems. A customized operating system provides precisely the functionality needed by an application and consumes up to four times less memory than other IoT operating systems without sacrificing performance.**

Keywords—IoT, Operating System, Design and Implementation

I. INTRODUCTION

At the dawn of the 21st century, it had become possible to assemble small, lower-powered devices that combined computing and sensing. These early sensor network systems supported applications that ranged from wildlife tracking [19] to monitoring of the world's infrastructure [18].

A. *From Wireless Sensor Networks to Internet of Things*

Early wireless sensor networks (WSNs) were composed of tiny sensor devices with wireless communication capabilities. Applications running on these devices were simple, because their sole purpose was to take measurements of the physical world and transmit data back to a server [13]. However, due to limited computing power and memory, these devices could not run commodity operating systems, such as Windows® and Linux, so researchers created embedded systems, such as TinyOS [8] and Contiki [2], on which developers could deploy sensor applications on these first-generation devices.

These devices used non-commodity communication protocols, such as ZigBee and Z-Wave, tailored to their communication hardware. As devices' capabilities increased, they became able to make use of more general-purpose protocols, such as 6LoWPAN, thus increasing device interoperability [16]. For example, today, a light sensor device can ask lamp controllers to make a room brighter or darker, based on the brightness level it measures. These more capable devices are interconnected and jointly create the network of physical objects, called the Internet of Things (IoT) [10].

The IoT has attracted a lot of attention, because tiny devices play an important role in smart city management [1], healthcare [12], home automation [15], etc. Their applications are more sophisticated than previous sensor ones, demanding efficient memory management, multitasking, and real-time operations. For example, when a wearable electrocardiogram device detects an irregular heart rhythm, it is vital to alert both the patient and the physician in a timely fashion [17]. These second-generation devices frequently run newer, real-time operating systems, such as FreeRTOS [22] and Zephyr [23], to address those demands.

B. *Key Challenge*

To date, the IoT has had a tremendous impact in applications ranging from healthcare to home appliances, but there remains much potential. As the class of applications for IoT devices expands, each generation is likely to impose ever-increasing demands on the software infrastructure, so how should we build system software to deal with these wildly different application requirements on resource-constrained devices?

Building a unified general-purpose operating system is one solution but not necessarily the best one, because such a system will always have features that particular applications do not need. Further, they might not provide precisely the right behavior that an application expects. For example, there are two common execution models, thread-based and event-driven. Controller devices that wait for commands and trigger actuators are easier to express in an event-driven model (§11.3.1, §11.3.2), while gateway devices that translate messages from one protocol to another concurrently are better implemented using a thread-based model (§11.3.3). However, operating system designers typically make the choice on behalf of developers, thus exposing abstractions that may not be suitable for particular applications.

C. *Our approach and motivation*

We claim that the solution lies in making it easy to develop application-specific IoT operating systems to meet these rapidly increasing and diverging demands. However, building a special-purpose system from scratch is overwhelmingly burdensome, so we introduce Tinkertoy, a collection of standard modules from which one can assemble a custom operating system in only a few lines of code. We are inspired by the success of the Unikernel [14] and the design concept of library operating systems [3]. In the same way that Unikernels let developers select only those libraries needed by an application, Tinkertoy lets developers

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

select a set of modules from which to assemble a custom system that provides precisely the functionality needed by an application. As such, we answer the following three questions:

- What are the common modules needed in IoT system software?
- How does one construct standard operating system modules so that applications can easily mix and match?
- How do we build such modules so that the assembled systems do not suffer high runtime overhead?

D. *Odyssey to Tinkertoy*

While Unikernels are motivated largely by application-level resource management and removal of protection boundaries, our goal is to assemble customized operating systems that have a small memory footprint and runtime performance comparable to that of other IoT operating systems. Overall, this work makes the following contributions:

- We present Tinkertoy, a set of standard modules that can be assembled into a custom IoT operating system (Sections IV through IX).
- We exploit recent C++ language features that make each module as flexible as possible while still allowing for their efficient implementation (Section III).
- We show that the effort of assembling a custom kernel is insignificant in terms of the number of lines of source code (Section XI.C).
- We show through an empirical case-based study that assembled kernels have a smaller memory footprint and better runtime performance than other popular IoT operating systems (Section XI.D).

II. Architecture Overview

A. *Target Devices*

Tinkertoy is composed of the 10 modules, shown in Table 1, from which developers assemble kernels for *low-end devices*, as classified by IETF [21]. Such devices have a single-core ARM Cortex-M processor without an MMU, limited memory, sensors and/or actuators, and communication hardware to interact with other devices. However, Tinkertoy is not restricted to support only this kind of device, because most modules are, in fact, general-purpose. As more modules become available, we imagine that developers can assemble kernels for devices, for example, that have multiple symmetric cores.

B. *Overview of Module Interactions*

Before digging into the details of each module, we illustrate how they interact with each other to provide services to user applications. Consider a system consisting of two tasks in which a watchdog task monitors a worker task and restarts it upon abnormal termination. Fig. 1 depicts the watchdog task making a system call to wait and turn control over to the worker task.

When the system boots, the kernel initializes itself and lets the watchdog task run (1). The watchdog task invokes a system call to wait until the worker task finishes (2). The system call raises an exception, causing the processor to switch to privileged mode and jump to a predefined kernel entry point in the context switcher (3). The context switcher preserves machine execution state on the watchdog task stack and restores the kernel state from the kernel stack (4), after which it returns to the dispatcher. The dispatcher relies on two companion components to process the request (5); it calls the service identifier finder to retrieve a unique service identifier (6) that is subsequently needed by the service routine mapper to select the kernel service routine (7) that implements the system call *wait()*. The routine receives a reference to the watchdog task (8) and finds that the task should be blocked, so it asks the scheduler to dequeue the worker task (9) which is then returned to the dispatcher (10). The dispatcher knows the next task to run, so it asks the context switcher to exit from the kernel, switch back to the unprivileged mode (11) and resume the worker task (12). Although not used in this example, there are multiple memory allocators that provide dynamically allocated memory for the kernel and user applications as needed.

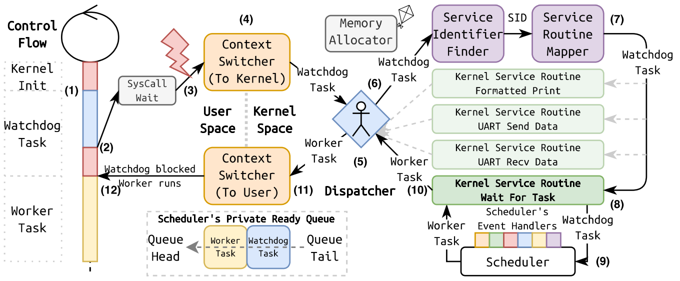

Fig. 1. Illustration of interactions between modules to service user requests.

TABLE I. Tinkertoy Modules

Modules	**Component Availability**[a]		**Related Sections**
	Prebuilt	***Building Blocks***	
Constraints	Provided by Tinkertoy		III
Scheduler	✓	✓	IV
Memory Allocator	✓	✓	V
Context Switcher	✓		VI
Execution State	✓		VI
System Call	✓		VI
Dispatcher		✓	VII
Kernel Service Routines	✓	✓	VIII
Execution Models	✓	✓	IX
Task Control Block		✓	IX

[a.] A component can be a prebuilt one, assembled from building blocks or implemented by developers.

C. *Overview of Module Decomposition*

Most of the above modules can be found in a conventional operating system, but Tinkertoy provides them as configurable building blocks. Specifically, some modules (e.g., the scheduler) are divided into components, allowing developers to customize the behavior of a module by switching one of its components for another prebuilt one, assembling a component from our building blocks, or using their own implementation. For example, one could replace the FIFO queue with a priority queue to make a scheduler prioritized. Fig. 2 depicts the relationship between a customized kernel, modules, components and building blocks.

Tinkertoy provides building blocks for both thread-based and event-driven models, so developers can customize the contents of the task control block, define kernel service routines that implement system calls and service hardware interrupts, and expose related system calls to assemble an execution model that best suits their applications. While offering great flexibility to developers, Tinkertoy uses constraints (details in §3.1) to ensure that other kernel modules are independent of the execution model and that developer-specified modules can be reasonably assembled into a kernel. Currently, Tinkertoy limits such kernels to being single-threaded and non-reentrant, so it does not yet support nested hardware interrupts and multiple kernel stacks. Nevertheless, our building blocks are not designed under these assumptions and can be combined with multithreading-specific ones, e.g., to protect data structures used by a memory allocator.

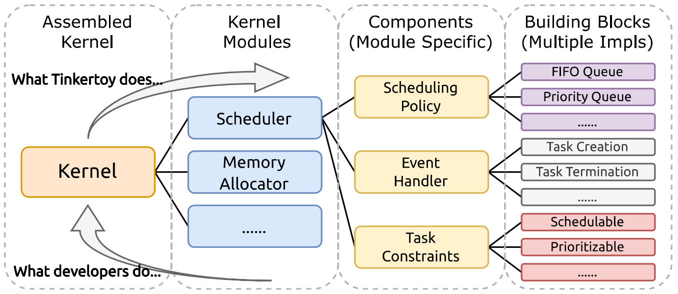

Fig. 2. Illustration of decomposing a kernel into primitive building blocks that can have multiple implementations to suit different applications' needs. Developers assemble a kernel by reversing the decomposition process.

III. Methodology

Our goal is to create flexible building blocks for kernel modules without sacrificing runtime performance. We introduce the following three key design principles and the corresponding set of C++ language features we use to support them.

A. Constrained Flexibility

We use *templates* to capture the genericity of kernel modules whose mechanisms are, in fact, independent of other modules. For example, on some systems, a scheduler schedules processes, while on others, it schedules threads, so its design should be agnostic to the object to schedule, but a specific instance of a scheduler should be constrained to schedule objects of a specific type. For example, a priority-based scheduler must know how to reorder tasks by their priority level, so we express constraints on the task type using *concepts* standardized in C++20 [25].

```
template <typename T>
concept Comparable = requires(const T& lhs, const T& rhs)
{  { lhs < rhs } -> std::same_as<bool>;  };
template <typename Task> requires Comparable<Task>
struct MyPriorityScheduler {};
```

Fig. 3. Definition of a concept that requires a type to overload the *less than* operator. *MyPriorityScheduler* can be specialized only if *Task* is *comparable*.

A *concept* allows one to define a set of requirements on a type, such as which member functions the type must implement, which operators the type must overload, and which member types the type must define. A *template* can be associated with one or more *concepts*, jointly imposing constraints on the template parameter. When specializing a template, developers must provide a concrete type that satisfies all the requirements specified by those *concepts*, otherwise the compiler will not compile the code. Tinkertoy's constraints are designed to be as concise and tight as possible but admit a large space of potential instances that satisfy the type requirements. For example, a priority-based scheduler can accept any task type that overloads the comparison operators. Fig. 3 shows an example concept definition and its usage.

B. Code Reusability

Developers should be able to use existing building blocks to assemble different instances of a particular kernel module, so it is imperative to make building blocks highly reusable. For example, any preemptive scheduler preempts a running task by placing it on the ready queue and resuming the next ready task. As such, we encapsulate the functionality of a building block as a *functor*, which is a C++ class that overloads the function call operator. In comparison to a C-style function pointer, a functor has all the benefits of class; it can be stateful, generic, and constrained by *concepts*, but more importantly, the compiler can inline calls to the functor, providing better performance than making an indirect function call at runtime. Fig. 4 shows an example functor that takes no arguments and returns void.

C. Code Composability

Tinkertoy provides a collection of standard building blocks for each kernel module, but developers might need to extend the functionality of an existing building block or combine multiple ones to satisfy their needs. For example, a multilevel feedback queue scheduler needs to allocate a quantum to a task before placing it on the ready queue. Subclassing and delegation are two common approaches, but they both incur runtime overhead, for example, by using virtual functions.

Since building blocks are encapsulated as functors, we use *fold expression* from C++17 to create a new building block from existing ones at compile time. A fold expression is used with a template parameter pack, so Tinkertoy provides builder classes, covered in the remaining sections, taking building blocks as their template parameters and producing a new block. Fig. 4 shows an example of creating a new functor from existing ones.

```
template <typename... Functor>
struct FunctorBuilder
{  void operator()() { (Functor{}(), ...); }  };
using MyFunctor = FunctorBuilder<Functor1, Functor3, Functor2>;
MyFunctor{}();  // Print "132"
```

Fig. 4. Implementation of a builder class that uses a *fold expression* to invoke, in the specified order, an arbitrary number of stateless functors, each of which is assumed to return *void* (for simplicity). Note that the builder class itself, when specialized, is a functor. Each *FunctorX* prints the digit "X", so the assembled functor, *MyFunctor*, prints "132".

D. Summary of Methodology

Tinkertoy leverages recent C++ language features to ensure that building blocks are reasonably flexible, highly reusable, and easily composable. These features allow us to write concise and efficient code. While we use C++, other languages have similar features, e.g., Rust's *type traits* mechanism allows one to enforce constraints on a type at compile time, providing functionality similar to that of C++ *concepts*. We leave exploration of designing and implementing building blocks in other languages for future work.

IV. Scheduler

A scheduler decides which task should run and for how long it should execute. Its policy can be classified along different dimensions, such as preemptive vs. cooperative or prioritized vs.

non-prioritized. Tinkertoy's scheduler is composed of three components: Policy, Event Handlers, and Task Control Block Constraints. We analyze popular scheduling algorithms and model their commonalities as a set of building blocks shown in Table 2. After explaining this decomposition, we illustrate how to assemble a scheduler to meet application needs.

TABLE II. SCHEDULER BUILDING BLOCKS

Scheduling Policies	**Event Handlers**
First In First Out Queue	Timer Interrupt
Prioritized Single Queue	Task Creation
Prioritized Multi Queue	Task Termination
Task Control Block Constraints	Task Yielded
Schedulable	Task Blocked
Implicitly Prioritizable	Task Unblocked
Prioritizable By Priority	Task Killed
Prioritizable By Mutable Priority	Task Priority Changed
Prioritizable By Auto Mutable Priority	Task Self Priority Changed
Quantizable	Task Quantum Used Up

A. *Module Decomposition*

A scheduler maintains one or more queues to keep track of ready tasks (1) and decides which task to run in response to external events (2), such as a new task arriving in the system (3). Fig. 5 depicts the interactions between the scheduler and other kernel modules.

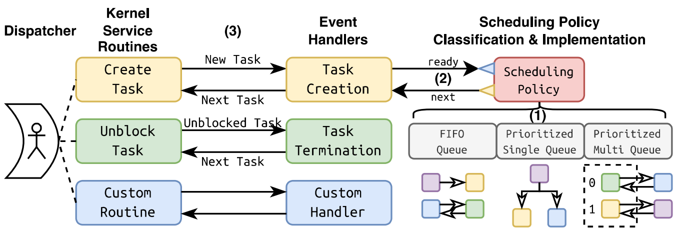

Fig. 5. Illustration of interactions between scheduler components.

1) *Policy Component*

The policy component manages the ready queue reflecting whether a scheduler is prioritized. It exposes two primitives, *ready* and *next*, to enqueue and dequeue a task respectively. The queue accepts tasks only if they are schedulable, so we define the class *Schedulable*, from which all schedulable tasks must inherit.

Tinkertoy provides three types of queues: FIFO, Prioritized Single, and Prioritized Multi. A scheduler that adopts a priority queue must assign priorities, so we define constraints, *Implicitly Prioritizable*, which requires a task to overload the comparison operators if it wants to keep its priority level private to the scheduler, and *Prioritizable by Priority*, which requires a task to reveal its priority level via a getter function. The actual type and the meaning of a priority level is determined by developers. For example, one can treat a task's deadline or periodicity as its priority level and assemble an Earliest Deadline First or Rate Monotonic scheduler for a real-time operating system. The Prioritized Multi Queue component allows developers to specify a potentially different policy for each priority level. A multilevel queue scheduler needs a developer-provided *mapper* (a functor) to initialize each queue. For example, one can build a Prioritized Round Robin scheduler by providing a mapper that returns a FIFO queue for each level.

Developers can build a new policy component from an existing one and a list of functors, as shown in Fig. 6, to customize enqueue and dequeue behaviors, such as updating the amount of time the task has been running. In general, the policy component makes it possible to materialize the queue for all common scheduling algorithms.

2) *Event Handler Component*

The event handler component reacts to scheduling events that can occur on a system and reflects characteristics such as whether a scheduler is preemptive. For example, when a new task is created, a cooperative scheduler might keep the current task running, while a preemptive one might run the task with the higher priority. These two different reactions are reflected in the implementation of the task creation handler.

Tinkertoy provides ten types of event handlers, listed in Table 2, allowing developers to specify to which types of events a scheduler can respond and how it should respond. Each of them defines the interface through which other kernel modules interact with the scheduler and uses primitives provided by the policy component to manipulate the ready queue. For example, if a task can change its priority at run time, the kernel service routine invokes the Priority Changed handler to inform the scheduler that the priority level of a task has been changed; it is then up to the scheduler to reschedule if necessary. Our design has no limit on the number of handler types, so developers are free to declare and implement new types of handlers.

B. *Building Custom Schedulers*

Tinkertoy provides a builder class to assemble a scheduler from a policy component and a collection of event handlers. We illustrate this process to assemble a simple FIFO scheduler.

1) *System Requirements*

Let's say that we want to build a scheduler for a kernel that allows a process to create another process, relinquish the processor voluntarily and wait for that process to finish. We assume the existence of a user process that never terminates (e.g., Unix's *init* process). The kernel expects all processes to run in arrival time order and wants to track the number of times a task is preempted. The system does not have a hardware timer.

2) *Assembling Schedulers*

The above requirements suggest that we need five event handlers: Creation, Termination, Yielded, Blocked and Unblocked. Since the system has a never-terminating user process, there is always a runnable process, so we do not need an idle task. We build a new policy component on top of FIFO that increments the preemption counter of a task to be enqueued. We assemble the FIFO scheduler as shown in Fig. 6.

```
using Policy = PolicyWithEnqueueExtensions<FIFO, Counter>;
class MyFIFOScheduler : public SchedulerAssembler<Policy,
    TaskCreation::Cooperative::KeepRunningCurrent<Task>,
    TaskTermination::Common::RunNext<Task>,
    TaskBlocked::Common::RunNext<Task>,
    TaskUnblocked::Cooperative::KeepRunningCurrent<Task>,
    TaskYielded::Common::RunNext<Task>> {}
```

Fig. 6. Composition of a FIFO scheduler using existing components.

V. MEMORY ALLOCATOR

Physical memory is a scarce resource on tiny devices without MMUs. Tinkertoy provides four prebuilt memory allocators, a free list allocator, a fixed-size resource allocator, a fast pool allocator, and a binary buddy allocator, as well as building blocks for developers to assemble custom allocators. We discuss how we decompose an allocator into components and show that our binary buddy allocator assembled for memory-constrained devices outperforms the Linux one in memory efficiency.

A. *Module Decomposition*

Dynamic memory allocation involves two major operations, *allocate* and *free*. We decompose a memory allocator into four components: Memory Block, Static Aligner, Primitive Steps, and Account Book. Fig. 7 depicts the interactions between them.

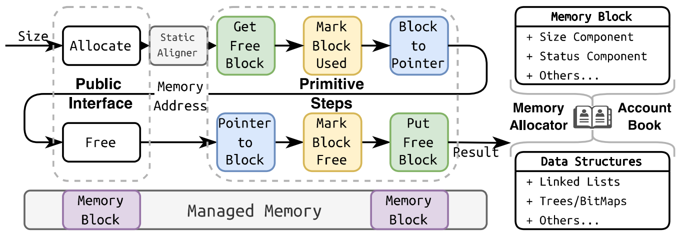

Fig. 7. Illustration of interactions between memory allocator components.

1) *Memory Block*

A memory block is an abstraction for a region of memory that might contain metadata necessary for the allocator to track space that is (un)available for allocation. For example, a free list allocator needs to know the size of each memory region, while a fixed-size one needs to know only whether a region is free. As a result, the contents of a memory block are allocator-specific.

Tinkertoy provides a collection of standard memory block components to store the size, track the allocation status, maintain pointers to next blocks, etc., allowing developers to assemble a custom memory block simply by selecting existing components. Alternatively, they can implement an allocator using some or none of the existing components. For example, if a device has less than 64 KB memory, one could use two 2-byte integers to store the block size and the address of the next free block.

2) *Static Aligner*

Variable-sized memory allocators might rely on an aligner (a functor) to ensure that all allocations are properly aligned to a fixed boundary. An aligner calculates the amount of memory needed to satisfy both the allocation request and the alignment requirement; it is invoked by *allocate*. Tinkertoy provides three types of aligners: null, constant and power-of-two.

3) *Primitive Steps*

Allocate is composed of three primitive steps, Get Free Block, Mark Block Used, and Block to Pointer. Get Free Block tries to find a free memory block large enough to satisfy the request. Subsequently, Mark Block Used might modify the metadata to mark the block in use. At the end, Block to Pointer returns the start address of the block to the program.

Similarly, *free* is also composed of three primitive steps, Pointer to Block, Mark Block Free, and Put Free Block, each of which does the reverse of the corresponding step in *allocate*. As such, we can provide a default implementation for both *allocate* and *free* with these primitives, while leaving developers the freedom to customize the behavior of each step.

4) *Account Book*

Primitive steps track allocations with specific data structures, each of which implements part of an allocation algorithm. Tinkertoy provides two types of account book, overlay and standalone. The former stores metadata of a memory block in the block itself, while the latter allocates additional memory for metadata. We also provide common data structures for each of them, such as a list and a binary tree.

B. *Memory Efficiency of Assembled Binary Buddy Allocator*

Tinkertoy's binary buddy allocator allows developers to specify a basic allocation size S and the maximum order N. It uses a standalone binary tree represented as a bit array to track the status of memory blocks of every possible size between S and $2^N S$, consuming only 2^{N-3} bytes. In contrast, Linux's binary buddy allocator uses an array of free lists, each of which is associated with a bit map that tracks the status of each pair of buddy blocks of a given order, thus reserving $16N$ bytes for N orders [6]. Fig. 8 shows the amount of memory reserved by Linux's allocator compared to that of Tinkertoy's.

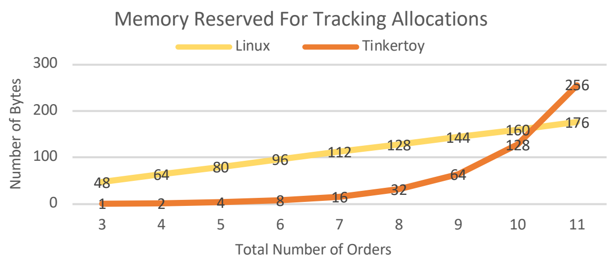

Fig. 8. A comparison of the amount of memory reserved, in bytes, for tracking memory allocations.

The results demonstrate both the memory efficiency of Tinkertoy's allocator for the common case, which has relatively few orders, and the advantages of Tinkertoy's design: A developer who needs more than ten orders can adopt the Linux strategy by replacing the binary tree with a linked list and extending the primitive steps of a free list allocator to update the status of each pair of buddy blocks.

VI. Context Switcher

Tinkertoy's context switcher is composed of two halves, *to-kernel*, which defines kernel entry points and switches from a user task to the kernel, and *from-kernel*, which does the reverse. The processor jumps to the to-kernel half on an interrupt, so the context switcher preserves the execution state and hands control to the dispatcher. After processing the request, the dispatcher calls the from-kernel half with the task that runs next. We model execution state as an architecture-dependent object, specifying the layout of saved registers and the calling conventions.

A. *Execution Model Independent Design*

A common place to store a task's execution state is its stack. After copying the machine's registers into stack space, the context switcher stores the address of the saved state into the task's control block, so it relies on constraints to have full access to a task's member field *stack pointer*, while remaining agnostic to whether a kernel is event-based or thread-based. Tinkertoy provides prebuilt context switchers for both x86 and ARMv7-M for non-reentrant, single-threaded kernels.

B. System Call and Execution State

Tinkertoy allows developers to define their own system calls, using a broker function, *syscall*, for each architecture. The broker is a variadic function that takes a numeric system call identifier followed by an arbitrary number of arguments and returns a signed 32-bit integer representing the kernel return value; all system calls are essentially wrappers of this function.

The interface between *syscall* and the kernel uses registers, passing the system call identifier in one register and a *va_list* pointer in another. However, *syscall* must accept its arguments from wrappers according to calling conventions. If the calling conventions specify register parameters, *syscall* uses those; if the calling conventions require arguments to be passed on the stack, it uses two caller-saved registers, instead, ensuring that the original values are preserved. The kernel should also interact with its service routines adhering to calling conventions. To facilitate this interaction in an architecture-independent fashion, we define a constraint on the execution state to ensure that it specifies the calling conventions for system calls by providing the following functions for kernel service routines in Fig. 9.

```
template <typename C>
concept ContextSpecifiesSysCallConvention = requires(C& c, Int32 v)
{
    { c.getSyscallIdentifier() } -> std::same_as<UInt32>;
    { c.getSyscallArgumentList() } -> std::same_as<va_list*>;
    { c.setSyscallKernelReturnValue(v) } -> std::same_as<void>;
};
```

Fig. 9. Definition of the constraint on the execution context to specify the calling convention of system calls.

As a result, the implementation of the broker function is paired with that of the execution state. For example, Tinkertoy loads the system call identifier into *%r0* on ARMv7-M, so it implements the function *getSyscallID* by returning the register value of *%r0* in the saved execution state. Developers do not need to provide their own implementations of both components unless they port Tinkertoy to another architecture.

VII. Dispatcher

The dispatcher connects user tasks and devices that request services and kernel service routines that provide services. It does not implement any kernel policy and is independent of the execution model. We divide the dispatcher into two halves, the *specific* half, which invokes kernel service routines requested by a task or a piece of hardware, and the *common* half, which invokes routines shared by all tasks, such as checking pending signals.

A. Companion Components

After receiving a service request from the context switcher, the dispatcher relies on two developer-specified components, each of which is a functor, to select a kernel service routine that can process the request. The Service Identifier Finder takes a pointer to the interrupted task and returns the service identifier. For example, ARM Cortex-M processors have a special Interrupt Control and State Register (ICSR) that records the current IRQ number, which can be used as the service identifier. The Service Routine Mapper consumes a service identifier and returns a pointer to the service routine. Since IRQ numbers are fixed on ARM Cortex-M systems, developers can use either a table or a switch statement to implement the mapper.

B. Assembling Dispatchers

Tinkertoy provides a builder class for assembling custom dispatchers. Consider an ARM Cortex-M3 system that provides two system calls, *syssend* and *sysrecv*, to send or receive data via a UART port. The system uses the IRQ number as the service identifier and supports signal delivery from the kernel. As such, we assemble the dispatcher as shown in Fig. 10.

```
using MyDispatcher = Dispatcher<TCB, UInt32, MyIdentifierFinder,
MyRoutineMapper, ContextSwitcherARM, SetupSignalHandlerContext>;
```

Fig. 10. Assemble a custom dispatcher for the system.

The service identifier finder reads the 32-bit IRQ number from the ICSR register. If the number is 11, the mapper tells the dispatcher to redirect the request to the kernel service routine that processes system calls. Similarly, if the number is 22, the request is redirected to the UART RX interrupt handler. Since the kernel can send signals to a thread, we specify the special service routine *Setup Signal Handler Context* that builds the execution state for the thread that runs next if a signal is pending.

VIII. Kernel Service Routines

Kernel service routines implement system calls and respond to hardware interrupts. Depending upon the type of request, they might ask the scheduler to reorder tasks and/or dequeue the next available task. Additionally, they rely heavily on constraints and thus provide services only if all requirements are satisfied. As such, kernel service routines have the following three properties.

A. Non-blocking

Recall that Tinkertoy kernels are currently single-threaded, so all kernel service routines must be one-shot and run to completion. However, developers do not need to write routines in a continuation passing style, such as having a callback parameter in the function signature, because a service routine that must be blocked waiting for some resources or conditions in a multithreaded kernel can be naturally expressed as two (or more) non-blocking service routines in a single-threaded kernel.

For example, when a task requests a read from a serial port, the service routine that implements *sysrecv* checks whether the kernel buffer has enough data to satisfy the request. If so, it copies the data and asks the dispatcher to resume the task. Otherwise, it places the task on the waiting queue of the driver and retrieves the next task from the Task Blocked event handler of the scheduler. Later, the serial device generates an interrupt to notify the kernel that data is available for reading, so the UART Rx interrupt handler is invoked to service the interrupt, moving data from the hardware buffer to the kernel buffer. When the kernel buffer has enough data to satisfy the receive request, the interrupt handler dequeues the receiver task from the waiting queue and asks the scheduler to unblock the task via the Task Unblocked event handler. Depending upon the scheduling policy, the scheduler might preempt the task being interrupted and resume the receiver task. Subsequently, the receiver task returns from the system call and proceeds with the data.

B. Task Control Block-Independent

Recall that Tinkertoy allows developers to assemble a task control block, so kernel service routines should be independent of any specific task control block type. However, a service routine might rely on certain task control block components (§9.1) to provide services. For example, the one that implements *sysgetpid* must be able to retrieve the identifier of the task that issues the request, so it uses the constraint *Has Unique Identifier* to guarantee read access to the identifier and *Can Invoke SysCall* to guarantee write access to the kernel return value. As a result, it is agnostic to how the identifier is stored in the task control block and which register will hold the kernel return value.

C. *Component-Independent*

We use a strategy similar to that described in the previous section to make kernel service routines independent of other kernel modules, such as the scheduler. For example, the service routine that implements *sysyield* must have access to the scheduler, which provides the Task Yielded event handler, while the one implementing *sysrecv* needs the Task Blocked event handler. As a result, we also translate these requirements into scheduler constraints, ensuring that kernel service routines have access to what exactly they need.

IX. Execution Models

Thread-based and event-driven models are two common execution models, but the debate over which one is better has raged for decades [7]. In reality, some systems are simply easier to express in one or the other model. Regardless of which model developers choose for their applications, the kernel maintains a task control block for each abstract unit of execution, such as a process, a thread, an event handler, or a coroutine.

A task control block should contain only the information needed by the kernel to provide services. For example, the kernel needs the priority level of each task to provide priority-based scheduling but does not need the task identifier if there is no system call that references a task by its identifier. As such, Tinkertoy provides building blocks from which to assemble, initialize and finalize a task control block and execution model.

A. *Task Control Block Components*

1) *Components "Meet" Constraints*

Tinkertoy provides 10 task control block components, each of which comprises a part of the final task control block by defining zero or more instance variables and providing functions to manipulate those instance variables, but more importantly, each task control block component can be used to satisfy a particular constraint as listed in Table 3.

2) *Stack Components*

Tinkertoy is designed to support a variety of different application architectures as efficiently as possible. For example, monitoring systems (§11.3.1) typically require a single task that loops infinitely collecting and transmitting data, while server systems, such as gateways (§11.3.3), use an endless supply of short-lived threads. These different application architectures impose different requirements on the kernel. Specifically, the kernel need never reclaim stack space if task lifetimes are essentially forever. Such tasks are best implemented by the non-recyclable stack component that keeps track of the current stack pointer only. In contrast, applications that use short-lived tasks should be implemented by the recyclable stack component that also records the start address of the stack, so the kernel can reclaim stack space for new tasks. However, developers do not have to limit themselves to these stack components; they can design their own, for example, one that avoids recording the start address of the stack by allocating a contiguous block of memory for both the task control block and the stack, as done in Linux.

3) *System Call Support Component*

Recall that kernel service routines that implement system calls read arguments and set the kernel return value (§8.2) by manipulating saved execution state (§6.2). To read a task's saved state, a service routine must be able to access its stack pointer. Since the execution state component provides an extra layer of indirection to hide architecture-specific details, we provide the system call support component for kernel service routines to access arguments and the return value conveniently. We use static polymorphism to explicitly make the system call support component dependent on one of the stack components.

4) *Other Components*

We also provide standard components to declare a task identifier, assign a priority level, adjust the task state, etc., but developers can always implement their own to provide more efficient application-specific memory management. Suppose that a system supports at most four dynamic priority levels and 16 tasks, one might prefer to use 4-bit task identifiers, 2-bit priority levels, and a 2-bit state representation, consuming only a single byte instead of requiring three 4-byte integers. Fig. 11 presents an example of assembling such a task control block.

TABLE III. Task Control Block Components & Constraints

Task Control Block Components	**Initializer Components**	**Satisfied Constraints**
Shared Stack	Assign Shared Stack	Has Stack
Dedicated Non-Recyclable Stack	Allocate/Assign Dedicated Non-Recyclable Stack	Has Dedicated Stack *Inherited From Has Stack*
Dedicated Recyclable Stack	Allocate/Assign Recyclable Stack	Has Recyclable Stack *Inherited From Has Dedicated Stack*
System Call Support	Setup Execution State	Can Invoke SysCall
Numeric Identifier (with/without field)	Assign Unique ID	Has Unique Identifier
Priority Level (with/without field)	Assign Priority	Prioritizable (Scheduler)
State (with/without field)	Assign Task State	Has Explicit State

B. *Task Control Block Initializers and Finalizers*

When the kernel creates a new task, it must allocate and initialize a task control block. Similarly, when a task finishes running or is killed by other tasks, the kernel must finalize and release its control block. Both operations are implemented as kernel service routines that rely on a task controller to allocate and release control blocks, leaving developers responsible for the allocation algorithm; they can, of course, use existing building blocks from the Memory Allocator (§5.1) to assemble a custom control block allocator.

Since developers assemble task control blocks from a collection of components, they should be able to assemble the corresponding initializers and finalizers as well. Tinkertoy provides one or more initializer components for each task control block component and convenient builder classes to assemble a custom initializer and materialize a kernel service routine. The same concept also applies to the finalizer, and as a result, developers can initialize or finalize a task control block in one line of code. Fig. 11 presents an example of initializing an assembled task control block.

```
struct Block: Schedulable, Listable<Block>,
  UniqueIDNoDecl<Block>, PriorityNoDecl<Block>, StateNoDecl<Block>,
  DedicatedNonRecStack<Block>, SysCallSupport<Block, Context>
{ UInt8 identifier: 4, priority: 2, state: 2; };
using Init = Builder<Block, AssignUniqueID<Block>,
    AssignPriority<Block>, AllocateDedicatedStack<Block>,
    SetupExecutionContext<Block, ExecutionContextBuilder>>;
Init{}(block, 1, kHigh, kDefaultStackSize, task_main);
```

Fig. 11. Assemble an initializer to initialize a task control block. The 1st argument block is the target to be initialized. The 2nd argument, with value 1, is the task identifier, and is passed to the initializer component Assign Unique Identifier. The rest arguments are passed to their corresponding components.

C. *Thread-based Execution Model*

Once developers have defined the task control block type and built the initializer and finalizer, they can expose relevant system calls, such as creating a new thread, to user programs in two steps. First, they declare the system call prototypes, assign a unique service identifier to each, and pass the identifier and arguments to the *syscall* function (§6.2). Second, they add an entry to the service routine mapper (§7.1), so that the dispatcher can route the request to the corresponding kernel service routine. Fig. 12 shows an example of implementing a system call *sysCreateThread* and routing the request to the task control block initializer built in Fig. 11.

Tinkertoy does not yet provide synchronization primitives, such as mutex, semaphores and message queues, so there are limitations in the current thread-based model (e.g., threads cannot communicate). However, our builder classes are generic enough to work with any number and type of components, so we believe that it is possible to implement such functionality.

```
int sysCreateThread(int id, int pri, size_t size, void* ep)
{ return syscall(kSysIDCreateThread, id, pri, size, ep); }
using Routine = ServiceRoutineBuilder<Block, Scheduler,
    ThreadController, AssignUniqueID<Block>,
    AssignPriority<Block>, AllocateDedicatedStack<Block>,
    SetupExecutionContext<Block, ExecutionContextBuilder>>;
OSDefineAndRouteKernelRoutine(kCreateThrd, Block, Routine);
```

Fig. 12. Add a new system call to create a thread at runtime. The Service Routine Builder materializes a kernel service routine for the new system call. Internally, it reads system call arguments and forwards them to initializer components. The kernel service routine is encapsulated as a functor, so the macro is needed to convert the functor to a function pointer which can then be used by the service routine mapper to route the request properly.

D. *Event-driven Execution Model*

The event-driven execution model allows developers to define custom events and model their applications as individual event handlers that run on a single shared stack. Tinkertoy provides task control components specific to this model, such as the event handler component that stores the handler entry point, and standard kernel service routines as well as system calls to (un)register events and their handlers and deliver events.

X. Related Work

Tinkertoy's building blocks make it possible to assemble a custom kernel that is inherently modular. Before evaluating the performance of the kernels we've assembled, we discuss how this work draws on work in library operating systems, the Exokernel [3], and the Flux OSKit [4]. We then introduce five popular IoT operating systems and compare them to Tinkertoy.

A. *Library Operating Systems*

General-purpose operating systems must multiplex hardware resources among user applications, so kernel designers select a policy for sharing each resource on behalf of users and expose a collection of abstractions to them. When applications impose wildly different demands, arbitrating among them becomes a key challenge. Library operating systems [14] address this concern by allowing developers to implement application-specific operating system abstractions in user space.

Exokernels [3] focus on presenting the right abstractions to directly expose hardware, allowing applications to manage resources efficiently at user level while still protecting them from each other. However, the kernel still defines core functionality and policies from which a library system can use or extend for a particular application. Tinkertoy allows developers to create or choose abstractions as well by providing kernel service routines and exposing related system calls, but it is a set of modules that are independent of the kernel architecture and execution model, so developers are free to replace, add, or remove any of them. Application-level resource management is not our goal, because we believe that developers can use hardware resources efficiently by carefully assembling a custom operating system from building blocks.

The Flux OSKit [4] focuses on reusing components from different existing systems to construct new ones. Its encapsulation technique makes it easier to import code, such as device drivers and network stacks, from other systems, and its object-oriented design makes it possible to mix and match different components to produce a new system. The Flux OSKit provides two modes of system construction: separable components and individual functions. Tinkertoy shares the design concept of modularity and separability with the Flux OSKit but does not import components from other operating systems, primarily because such components are frequently too large and expensive for IoT systems, but also because even the overhead of wrapping these components in standard interfaces might be too burdensome for our target environment. Instead, we carefully design each component from scratch to provide a third approach to system construction: composable code, so developers can customize kernel data structures (e.g., task control block) and related functions (e.g., initializers, finalizers, and service routines) and construct entirely new systems by assembling them out of a collection of modular, flexible, and interacting components.

B. *Other IoT Operating Systems*

TinyOS [8] is an event-driven operating system written in NesC [5]. Software services and hardware resources are encapsulated as components, interacting with each other by posting and responding to events. TinyOS achieves modularity at the component level; developers write applications in terms of a set of components but cannot customize a component easily without tweaking the code. Besides, TinyOS does not support user space nor dynamic memory allocation, making it vulnerable to faulty user applications and infeasible to accommodate unpredictable runtime resource requests. Its Rust-based successor, Tock [9], exploits both hardware and programming language protection mechanisms, focusing on fault isolation and efficient and safe memory management.

Contiki [2] is another event-driven operating system written in C. User applications are built upon the stack-less protothread library, so they run cooperatively and can only be preempted by hardware interrupts. The system supports dynamic memory allocation but not user space and is modular in a coarse-grained manner similar to TinyOS. Developers can edit a Makefile to include only the modules they need but still cannot customize the behavior of each module and the kernel easily like Tinkertoy.

FreeRTOS [22] is a real-time thread-based operating system written in C. Its kernel is designed to be small and simple and consists of only three source files: list, queue, and task. All other kernel functionality, such as semaphores and event groups, are built on top of these three constructs. Applications are encapsulated as tasks with a private stack and can be scheduled either preemptively or cooperatively. FreeRTOS relies heavily on C macros to make the kernel modular at the API level but not at the code level as Tinkertoy does. Developers customize the system by providing a header file in which they define macros to enable or disable kernel APIs.

Zephyr [23] is a thread-based operating system written in C. Its kernel shares many design concepts with Linux. For example, Zephyr provides multiple scheduling algorithms, such as cooperative, preemptive, time slicing and earliest deadline first, to suit application needs. It supports meta IRQ scheduling to defer executing a function, which behaves similarly to Linux's tasklet and softirq. Zephyr provides a Kconfig interface [24], as Linux does, allowing one to customize the system in a finer-grained manner than FreeRTOS does. Developers can specify the device drivers they need, the scheduling algorithm, the data structure that implements the queue, etc., and similar to Tinkertoy, some of the customizations are close to code level.

Using the following criteria, we select FreeRTOS and Zephyr as our baselines in the evaluation. First, these systems allow us to disable functionality that is not used, so both the baseline kernels and our kernel are doing the same work. Second, they are under active development and therefore do not present performance or implementation challenges in the kernel. Third, both have an official port that supports our emulated board, so we can focus on adopting different kernel APIs when porting applications to these systems, leaving platform-specific code intact.

C. Summary

Tinkertoy shares the goal of building application-specific operating systems with Exokernel and the Flux OSKit, but its scale and design target are considerably different. While other aforementioned IoT operating systems have design targets closely aligned with those of Tinkertoy, they provide a prebuilt kernel, from which developers remove unneeded functionality. In contrast, Tinkertoy is not a kernel but a set of modules, from which developers assemble a custom kernel, and thus it is the granularity at which one can customize kernel modules easily and design new modules specific to a particular application that makes Tinkertoy distinctive. Our design principles ensure that the unit of modification is as small as necessary. For example, it takes 109 lines of code to implement the kernel interface that changes a task's priority in FreeRTOS, whereas one can implement a kernel service routine in fewer than 10 lines of code, simply notifying the scheduler through the Task Priority Changed handlers, which can be reused by other routines, e.g., the one that prevents priority inversion in a multithreaded kernel.

XI. Evaluation

We have identified three different kinds of kernels that apply to many different IoT applications: 1) a monitoring kernel used for applications responsible for collecting sensor data, 2) an actuator kernel that is representative of modern cyber-physical systems in which an actuator changes its physical surroundings, and 3) a gateway kernel found in network-focused applications. We demonstrate that Tinkertoy naturally supports all of these kernels by assembling an instance of each type of kernel and combining them into a concrete use case: an automated watering system. These three kernels are chosen carefully to reflect first- and second-generation sensor network operating systems, and emerging third-generation systems that run on intelligent IoT devices [11] and have functionality closer to that of a general purpose operating system. We then analyze the memory footprint and the runtime performance of the assembled kernels and compare the results to other systems.

A. Automatic Watering System

An automatic watering system takes care of potted plants while people are away from their home. It is composed of three devices, a monitor, an actuator, and a gateway, each of which requires a custom kernel. The monitor periodically fetches the soil moisture level from a sensor placed in a pot. If the level drops below a user-defined threshold, it asks the actuator to start dripping water. Once the level reaches an upper threshold, it signals the actuator to stop. The actuator controls the gate of a water bottle. It opens the gate on receiving a dry alert from the monitor and closes the gate on a wet alert. Optionally, a gravity sensor can be placed in the bottle, enabling the actuator to notify the user when the bottle runs out of water by sending out a CoAP POST message to a Linux HTTP server via the gateway. The gateway device sits between the actuator and the server, acting as a transparent proxy that translates CoAP messages to HTTP messages and vice versa. Fig. 13 depicts the setup.

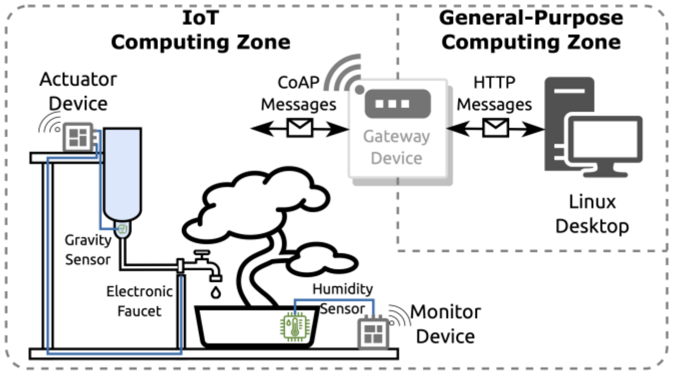

Fig. 13. Deployment of the automatic watering system in a home network.

B. Experimental Setup

We have three tiny devices in total, each of which runs its own custom kernel on a Stellaris LM3S811 board emulated by the ARM Fast Models. Each board has a 50 MHz ARM Cortex-

M3 processor, 64 KB Flash and 8 KB SRAM. Tinkertoy does not yet support networking, so these devices use UART ports to communicate with each other. We redirect each serial port to a TCP socket and run a controller process on the host machine to capture and display all messages within the network. Our emulated boards have sensors simulated in the kernel, whose values can be modified by the controller remotely. We use GCC 10.2.1 to compile all kernels with optimization level 3, code size optimization and link phase garbage collection enabled. C++ exceptions, runtime support and C standard library are disabled. As such, all unreferenced functions and variables are excluded from the final executable.

C. *Assembling Kernels*

We evaluate implementation effort in terms of the number of source lines of code (SLOC) to implement a kernel for each device. Application code is not counted despite being compiled with the kernel into a single binary image.

1) *Monitor Kernel*

System Requirements: The user application running on the monitor device acts as an interface between the moisture sensor and the water bottle and can be modeled as a state machine, so an event-driven kernel is adequate. Table 4 lists the three required events and their handlers. The system should deliver a timer event on a regular basis, so that the reader task can fetch and examine the moisture level and subsequently deliver a dry or wet soil event if necessary. The dry handler then sends a dry alert to the actuator, while the wet one does the opposite. Soil events are more critical than the timer one, so the kernel should run the handler immediately when an event is delivered.

TABLE IV. EVENTS IN MONITOR AND ACTUATOR KERNELS

Monitor Kernel		**Actuator Kernel**	
Events	***Event Handlers***	***Events***	***Event Handlers***
Periodic Timer	Sensor Reader	Start Watering	Open Gate
Dry Soil	Dry Handler	Stop Watering	Close Gate
Wet Soil	Wet Handler	No Water	Signal Alert

System Deployment: The above requirements suggest that developers need a preemptive scheduler that can respond to task creation and termination events. They also need the UART, SysTick and Simulated Sensor driver modules and provide these five items listed in Table 5 to assemble the monitor kernel.

TABLE V. ITEMS NEEDED TO ASSEMBLE THE MONITOR KERNEL

Item	**Description**
1	Assemble the event control block and the event controller.
2	Assign a unique identifier to each system call.
3	Define the service routine mapper for the dispatcher.
4	Provide a timer interrupt handler to deliver timer events.
5	Provide a startup routine to initialize all modules.

Since all event handlers share the same stack, an event control block needs to record only the handler address and pointers to other blocks in the scheduler's ready queue, thus consuming only 12 bytes. The event controller maintains a table of four event control blocks and arbitrates event registrations. The first entry is reserved for the idle event, while the rest are used by the application. The kernel provides five system calls to manipulate events, sensors, and serial ports, so a simple switch statement is sufficient to implement the mapper. The timer interrupt handler tracks elapsed time in ticks. Once the number of ticks reaches the threshold, it resets the counter and delivers a periodic timer event by means of notifying the scheduler that a new task is created. Tinkertoy provides a simple bootloader that sets up the kernel stack, initializes global variables and runs the kernel startup routine in ARM Handler mode. We need to initialize the interrupt vector table, the timer and serial ports, register event handlers, and start the dispatcher in the startup routine. As a result, we need 65 lines of code in total to assemble the monitor kernel. Table 6 shows the breakdown by component.

2) *Actuator Kernel*

System Requirements: The actuator kernel is similar to the monitor kernel, also requiring three events shown in Table 4. The UART RX interrupt handler receives messages from the monitor and delivers the first two types of events. Message handlers then open or close the water gate to start or stop watering the plant. If the water bottle is empty, the message handler delivers a *No Water* event, so that the *Signal Alert* handler will send an alert message to the gateway. However, the system must ensure that event handlers access the water gate in order. Specifically, the *Close Gate* handler cannot preempt the running *Open Gate* handler in case the moisture level changes before the actuator opens the gate.

Differences in Deployment: As such, the actuator kernel shares about 80% of its code of the monitor kernel. Developers need a cooperative scheduler instead of a preemptive one and implement the interrupt handler for the serial port instead of the timer. The actuator kernel has two more system calls to control the water gate and thus requires 90 lines of code in total, 52 of which are shared with the monitor kernel. We remove 17 lines of code (related to timer) and add 42 lines of code (related to serial) to finish assembling the kernel. Table 6 shows the per-component line counts.

3) *Gateway Kernel*

System Requirements: The gateway translates multiple CoAP or HTTP messages concurrently, so a thread-based execution model is necessary. The translator thread should finish its work without being interrupted, so that it can be ready to service the next message promptly. For demonstration purposes, there are at most three translators that can run concurrently. To reduce the complexity of detecting the size of an incoming message without a network stack, each CoAP request is assumed to be 32 bytes long.

System Deployment: The above requirements suggest a cooperative scheduler that can handle task creation and blocked and unblocked events. Task termination is not needed, because a translator thread works in an infinite loop, remaining blocked until it receives a message. Additionally, we need a memory allocator to allocate a private stack to each thread. As such, developers provide five items similar to those needed to assemble the monitor kernel, except that the timer interrupt handler is replaced by the serial one.

Since a thread never terminates, its stack is never released, so the thread control block records the current stack pointer and the status of the receive request, consuming only 24 bytes. The *status* stores a reference to the user buffer and the number of bytes requested by the thread and having been processed by the serial driver. The UART RX interrupt handler maintains a queue of threads blocked waiting for data. Once the request is fulfilled, the interrupt handler notifies the scheduler that the thread has been unblocked. Finally, we initialize the memory manager and other kernel modules in the startup routine. In total, we need 122 lines of code to assemble the gateway kernel. Table 6 summarizes the deployment result.

TABLE VI. DEPLOYMENT RESULT: BREAKDOWN BY COMPONENTS

Components	Monitor		Actuator		Gateway	
	Method	***SLOC***	***Method***	***SLOC***	***Method***	***SLOC***
Scheduler	ASM[b]	8	ASM	8	ASM	8
TCB	ASM	1	ASM	1	MIX[b]	29
Controller	ASM	1	ASM	1	ASM	14
Mapper	WFS[b]	21	WFS	23	WFS	17
Dispatcher	ASM	2	ASM	2	ASM	1
ISR	WFS	12	WFS	36	WFS	29
Init Tasks	WFS	7	WFS	7	MIX	13
Startup	WFS	13	WFS	12	WFS	11
Total	**65**		**90**		**122**	

b. ASM = Assemble from building blocks; WFS = Write from scratch; MIX = Both ASM and WFS.

D. *Comparison Results*

1) *Porting Applications to Target Systems*

Despite being a thread-based system, FreeRTOS provides a lightweight mechanism, *Event Group*, to write event-driven programs. Event handlers in the monitor and the actuator kernels are converted to threads that wait for and process events in an endless loop, while translator threads in the gateway kernel are ported as is. However, since there is no strict distinction between the kernel and the user space in FreeRTOS, system calls behave like ordinary library calls and thus require no context switches.

Zephyr is also a thread-based system, so we follow the same procedure as we did with FreeRTOS to port our applications, except that *Event Group* is replaced by *Binary Semaphore*. Zephyr can be configured to enable a user space, so the kernel checks system call arguments and performs a context switch; otherwise, system calls are the same as library calls. We choose to disable argument checking, because Tinkertoy does not have any protection against malicious arguments yet, and we did not want to impose an unfair disadvantage on Zephyr.

2) *Flash and Memory Footprint*

We use *readelf* to dump the size of each segment in the kernel executable and calculate the number of bytes reserved in Flash and main memory, excluding statically allocated kernel and user stack areas. Table 7 summarizes the result.

Tinkertoy significantly outperforms the other two systems, because our building blocks allow developers to include only entries that are needed in kernel data structures. For example, a task control block is 12 bytes in Tinkertoy's monitor kernel but 64 bytes in FreeRTOS and 112 bytes in Zephyr. Since there are four event handlers registered on the system, it is unsurprising that the other systems require more memory.

Zephyr additionally reserves about 1 KB to store kernel configuration parameters, such as the number of IRQs, device properties, and 344 bytes for the software ISR table. Even though we allocate threads and semaphores statically, Zephyr's kernel still requires memory management routines to allocate and release private data structure. However, the result is still reasonable, as the reference minimal Flash footprint with multithreading enabled is about 7 KB to 8 KB [20].

TABLE VII. FLASH AND MEMORY FOOTPRINT

Devices	Kernels	Flash Footprint (Bytes)		Memory Footprint (Bytes)	
		Raw	***Normalized***[c]	***Raw***	***Normalized***[c]
Monitor	Tinkertoy	2308	1.00	116	1.00
	FreeRTOS	4808	2.08	672	5.79
	Zephyr	11116	4.81	936	8.06
Actuator	Tinkertoy	2277	1.00	119	1.00
	FreeRTOS	4845	2.12	888	7.46
	Zephyr	8568	3.76	920	7.73
Gateway	Tinkertoy	3900	1.00	268	1.00
	FreeRTOS	5976	1.53	792	2.95
	Zephyr	10948	2.80	1032	3.85

c. Footprints are normalized to Tinkertoy.

3) *Active Stack Usage*

We trace the execution of each system and analyze their stack footprint. Table 8 presents the result.

TABLE VIII. ACTIVE STACK FOOTPRINT

Devices	Kernels	Active Stack Usage (Bytes)			
		Kernel	***User***	***Total***	***Normalized***
Monitor	Tinkertoy	84	208	292	1.00
	FreeRTOS	112	360	472	1.61
	Zephyr	176	592	768	2.63
Actuator	Tinkertoy	88	192	280	1.00
	FreeRTOS	116	412	528	1.88
	Zephyr	96	584	680	2.42
Gateway	Tinkertoy	88	1056	1144	1.00
	FreeRTOS	112	1160	1272	1.11
	Zephyr	96	1456	1552	1.35

Both comparison systems have a larger kernel stack footprint than Tinkertoy does, but the differences are insignificant in all but the monitor kernel on Zephyr, because the kernel runs periodic timer tasks on a shared system work queue instead of an ordinary thread. As a result, sensor reading is performed in the kernel and therefore increases the kernel stack footprint.

On the other hand, our monitor and actuator systems have a significantly lower user stack footprint than the others, because both are event-driven, and all the event handlers share a single user stack. The difference is reduced to about 10% in the thread-based gateway kernel, because translators on each system have the same implementation except for system calls.

4) *Performance*

Tinkertoy-based kernels provide precisely the functionality needed by applications, so they should have not only a low memory footprint but also more efficient runtime than other systems. We evaluate the performance by means of the amount of time it takes the gateway kernel to receive a CoAP request message and send out the translated HTTP message. Table 9 summarizes the statistics.

TABLE IX. GATEWAY KERNEL ROUND TRIP TIME IN MILLISECONDS

Kernels	Samples	Min	Max	Median	Mean	SD
Tinkertoy	1000	0.55	1.26	0.67	0.66	0.04
FreeRTOS		0.54	2.40	0.69	0.68	0.06
Zephyr		0.58	1.78	0.69	0.68	0.05

Our gateway system is comparable to FreeRTOS and Zephyr despite them having user space disabled. We analyze the assembly code and find that the difference is related to how the kernel reacts to interrupts and unblocks the translator thread.

When a new message arrives, the processor jumps to the UART RX interrupt handler directly on FreeRTOS. In contrast, Tinkertoy allows developers to select which functionality the kernel should provide, so the processor must traverse through the context switcher and the dispatcher (service identifier finder and the service routine mapper) before entering the interrupt handler, thus having a longer code path than FreeRTOS. Inside the interrupt handler, Tinkertoy essentially does the same thing as FreeRTOS, removing the translator thread from the waiting list and putting it on the scheduler's ready queue.

Meanwhile, Zephyr takes the multithreaded kernel into consideration; it relies on spinlocks to protect semaphores and the scheduler, thus introducing unnecessary barrier instructions for every list operation on a single-threaded system. However, Zephyr does not provide a separate implementation for such a system, so we learn that Tinkertoy must provide building blocks for both types of system in the future.

E. Summary

We evaluated Tinkertoy from three perspectives: the effort needed to assemble a custom kernel, memory footprint and runtime performance. We find that about 75% of the deployment code is straightforward and easy to write, as developers use our builder classes to concatenate a collection of components and materialize kernel services. The rest deals with device interrupts, but Tinkertoy does not yet provide standard building blocks in this area, so developers have to use existing kernel APIs to service the hardware manually at this moment.

The assembled kernels provide the exact functionality needed by particular applications, so Tinkertoy-based kernels have 1.5x to 4.7x smaller memory footprints than other popular IoT operating systems and comparable performance to them. However, there is still a large room for future improvement, such as standard API packages to improve applications' portability, source-to-source compilers to translate thread-based programs to event-driven ones and vice versa.

XII. Conclusion

Tinkertoy allows developers to assemble operating systems customized for particular applications in fewer than 150 lines of code by providing a set of configurable and composable components. We use constraints to ensure that each component is as flexible as possible while also ensuring that all the requirements are satisfied by developer-specified components. The assembled systems not only provide precisely the functionality that an application needs, but also reduce memory footprint by as much as a factor of four compared to existing IoT operating systems, with no performance penalty. While there are many avenues for future work, we demonstrated that it is possible to assemble realistic operating systems in this manner. Tinkertoy is an embodiment of this approach and demonstrates how to achieve flexibility and configurability to satisfy the requirements of heterogeneous applications.

Tinkertoy is available at https://github.com/0xTinkertoy.

Acknowledgment

We thank Dr. Reto Achermann, Chris Chen and Sid Agrawal, members of the Systopia Laboratory at University of British Columbia, for providing valuable feedback on this work.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] H. Arasteh et al., "IoT-based smart cities: A survey", IEEE International Conference on Environment and Electrical Engineering, 2016.

[2] A. Dunkels, B. Gronvall and T. Voigt, "Contiki a lightweight and flexible operating system for tiny networked sensors", 29th Annual IEEE International Conference on Local Computer Networks.

[3] D. Engler, M. Kaashoek and J. O'Toole, "Exokernel", Proceedings of the fifteenth ACM symposium on Operating systems principles, 1995.

[4] B. Ford et al., "The Flux OSKit", ACM SIGOPS Operating Systems Review, vol. 31, no. 5, pp. 38-51, 1997.

[5] D. Gay et al., "The nesC language: A holistic approach to networked embedded systems", Proceedings of Programming Language Design and Implementation (PLDI), June 2003.

[6] M. Gorman, Understanding the Linux Virtual Memory Manager. Upper Saddle River, N.J.: Prentice Hall PTR, 2004.

[7] H. Lauer and R. Needham, "On the duality of operating system structures", ACM SIGOPS, vol. 13, no. 2, pp. 3-19, 1979.

[8] P. Levis et al., "TinyOS: An Operating System for Sensor Networks", Ambient Intelligence, pp. 115-148, 2005.

[9] A. Levy et al., "Multiprogramming a 64kB Computer Safely and Efficiently", Proceedings of the 26th Symposium on Operating Systems Principles, 2017.

[10] L. Mainetti, L. Patrono and A. Vilei, "Evolution of wireless sensor networks towards the Internet of Things: A survey", 19th International Conference on Software, Telecommunications and Computer Networks, pp. 1-6, 2011.

[11] J. Mills, J. Hu and G. Min, "Communication-efficient Federated Learning for Wireless Edge Intelligence in IOT," IEEE Internet of Things Journal, vol. 7, no. 7, pp. 5986–5994, 2020.

[12] M. Perera, M. Halgamuge, R. Samarakody and A. Mohammad, "Internet of Things in Healthcare: A Survey of Telemedicine Systems Used for Elderly People", IoT in Healthcare and Ambient Assisted Living, pp. 69-88, 2021.

[13] D. Puccinelli and M. Haenggi, "Wireless sensor networks: applications and challenges of ubiquitous sensing", IEEE Circuits and Systems Magazine, vol. 5, no. 3, pp. 19-31, 2005.

[14] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. "EbbRT: a framework for building per-application library operating systems." In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation, 671–688, 2016.

[15] G. Toschi, L. Campos and C. Cugnasca, "Home automation networks: A survey", Computer Standards & Interfaces, vol. 50, pp. 42-54, 2017.

[16] J. Vasseur and A. Dunkels, Interconnecting smart objects with IP. Burlington, MA: Morgan Kaufmann Publishers/Elsevier, 2012.

[17] G. Xu, "IoT-Assisted ECG Monitoring Framework With Secure Data Transmission for Health Care Applications," in IEEE Access, vol. 8, pp. 74586-74594, 2020.

[18] N. Xu, "A survey of sensor network applications", IEEE communications magazine, 40(8), pp. 102-114, 2002.

[19] P. Zhang, C. Sadler, S. Lyon and M. Martonosi, "Hardware design experiences in ZebraNet", Proceedings of the 2nd international conference on Embedded networked sensor systems - SenSys '04, 2004.

[20] "Minimal footprint", Zephyr Project Documentation, [Online]. Available: https://tinyurl.com/zephyrminfp

[21] "RFC 7228: Terminology for Constrained-Node Networks", IETF RFC, [Online]. Available: https://tools.ietf.org/html/rfc7228

[22] "FreeRTOS", FreeRTOS. [Online]. Available: https://www.freertos.org/

[23] "Zephyr", Zephyr Project. [Online]. Available: https://zephyrproject.org/

[24] "Configuration System (Kconfig)", Zephyr Project Documentation, [Online]. Available: https://tinyurl.com/zephyrkconfig

[25] "Constraints and concepts", C++ Reference Manual. [Online]. Available: https://en.cppreference.com/w/cpp/language/constraints