INTRODUCTION

The digital transformation opportunity & the new paradigm

The IoT marketplace will expand twelve-fold between 2018 and 2023, exploding into a $195 billion opportunity. It’s no wonder that everyone from the smallest startups to Fortune 500s are jumping into the IoT platform sandbox.

The market is additionally flooded with limited point solutions and proprietary hardware that offer a short-term entry point, but that have the longer-term effect of fragmenting and diluting the market and weakening the actual opportunity.

The exponential growth of connected, intelligent devices calls for a new computing paradigm—not a rebrand and extension of legacy architectures.

These centralized, brittle solutions will quickly yield the market to decentralized, more robust architectures that drive long-term value across the entire IoT ecosystem.

The nio Platform is built to deliver this paradigm.

This white paper details nio’s technical architecture, and describes how its distributed system logic delivers a unique computing paradigm.

While nio fundamentally empowers systems of intelligence, we begin by explaining the details of a single nio instance to give the reader foundational understanding upon which to build. The focus then shifts to nio systems and their architectures.

This paper is intended for readers that already have a basic familiarity with nio and its elements.
# Table of Contents

**nio system basics and definitions** ................................................................. 4  
  hierarchical structure of nio ........................................................................... 4

**instances:** managing and executing distributed logic .................................... 5

**blocks:** configurable, reusable, and extensible functionality .......................... 6  
  the nio block development framework ............................................................ 7

**services:** infinite workflows of blocks ......................................................... 8  
  stream processing .......................................................................................... 8  
  easy configuration through block routing ...................................................... 9  
  service instantiation and lifecycle ................................................................ 10

**the core:** foundation of nio ........................................................................... 11  
  service manager ............................................................................................ 11  
  API for instance and service management .................................................. 12  
  additional management functionality - core components .......................... 12  
  instance interfaces and implementations - modules ..................................... 12

**communications:** elegantly orchestrating data interactions .......................... 13  
  service-to-service communications ............................................................... 13  
  service-to-service communications with Pubkeeper .................................... 13  
  service-to-service communication with blocks .......................................... 14  
  non-nio communications ............................................................................... 14  
  communication security: end-to-end encryption ......................................... 14

**systems:** putting it all together .................................................................... 15  
  distributed logic: vineyard use case .............................................................. 16
nio system basics and definitions

A nio system may be highly localized (as small as one machine) or widely distributed across many devices, diverse clouds, and distinct data centers. A nio system will have at least one instance of the nio Platform running in it. Systems comprise of compute, sensors, databases, user interfaces—any elements that interact regularly with a common purpose. Instances within the system are able to communicate with one another in real time. A generic system with component details is shown in Figure 1 below:

Figure 1: Structure of a nio system

Hierarchical structure of the nio Platform

1. **nio system** - A connected group of nio instances and other entities (e.g., devices and software) that work together.
2. **nio instance** - A running (instantiated) version of nio that manages and executes the programmable logic of services.
3. **nio service** - A runtime within an instance that determines how streaming data (signals) move between modular logic (blocks).
4. **blocks** - A stateless or stateful function that operates on streaming data within the runtime environment of a service.
5. **nio core** - The underlying infrastructure of nio that launches and supports nio services.
6. **Pubkeeper** - A communications broker that enables devices and interfaces that would otherwise be unable to easily link to communicate securely and seamlessly with a simple client.
instances: managing and executing distributed logic

A nio system will have at least one instance of the nio Platform running within it. nio instances manage and execute system logic. Figure 2 illustrates the internal interaction of the basic parts of a nio instance:

Figure 2: Functional relationships within a nio instance

The nio Platform has two states: a pre-launch first state, and a post-launch second state (i.e., prior to and following instantiation). When nio is running (the second state), it is commonly referred to as an “instance.” Figure 3 illustrates the two states of a nio.

A nio instance can be divided into processing functionality (services/blocks) and management functionality (core). The configurable parts of nio (e.g., the task specific functionality) are the blocks and services. We explore those two concepts in depth in the next two sections.
Blocks are packages of functionality that allow for seamlessly unlimited extensibility within the nio Platform. They are reusable, configurable, and easy to build. Blocks may leverage open-source libraries and be open-sourced themselves.

At runtime, each block is instantiated using (1) a block class and (2) configuration information:

1. Each block class extends a base block class with task-specific functionality. This task-specific functionality provides logic that executes when the block is called by any service to which the block is assigned. The base block class provides default logic that enables the block to operate within the runtime environment of a service. For example, a block can be extended from the base block class with logic to enable connections to social media accounts.

2. The configuration information allows parameters to be set for each instantiation of the block. In our social media block example, the configurable parameters could include a username and password, as well as API information needed to connect to a particular account.
The same block class may be instantiated multiple times by the same service or by different services. The use of different configuration information for different instantiations allows customization to occur at runtime using the same block class.

**Figure 7:** Different runtime configurations of a single block class

---

**THE NIO OPEN BLOCK DEVELOPMENT FRAMEWORK**

Using the nio Block Development Framework, developers can quickly build new blocks to test out an idea or connect a new piece of hardware. Developers can leverage existing libraries and frameworks in their blocks. For instance, building a machine learning block does not require thousands of lines of code or custom scripting. Instead, just import the TensorFlow library into your block class and call into the library from the block.

Encouraging open-source blocks also allows all nio users to benefit from the power of community. Not a day goes by where a new web service or hardware device is not released to the market. Keeping up with these trends and integrating new prototypes into existing projects is a tall order. With nio, including the latest widget simply means using a new block. No single person or company could possibly create blocks fast enough to keep up with all the new ideas that are released. But the developer community as a whole can.

---

Functioning block code that leverages the open source Requests library (http://docs.python-requests.org/en/master/) to make HTTP POST requests to a custom URL based on incoming signals.
services:
infinite workflows of blocks

nio services are a runtime framework within which blocks execute. They dictate data creation, movement, transformation, and application. Services are easily modified and replicable across instances.

At runtime, each service is instantiated using (1) a service class and (2) configuration information:

1. Although a service class can extend a base service class, the service class is usually identical to the base service class: it provides all needed functionality.

2. The configuration information provides the service with specific information needed at instantiation such as the blocks to be run by the service and the order of execution (e.g., the order in which the blocks are called).

---

STREAM PROCESSING

nio is a stream processing engine. Signals are always in motion in nio: moving from block to block and from instance to instance. nio does its work on this in-transit data—transforming, replicating, and routing it.

Put another way, nio does not need a database to function and create value. This is in direct contrast to many cloud IoT platforms. These tools typically necessitate data transmission from client to server, with observable latency, before any action is taken. Often data (noise) will be moved to a data lake never to be seen again except as increased expense on your monthly cloud bill.

Stream processing with nio encourages a healthy tendency toward action. It opens up significant application possibilities and real-time action at the edge, but does not prohibit the storage of data.

To persist data at rest for longer durations, you simply add the relevant nio Block to your service and start sending signals to your choice of database or a persistence layer.
Easy configuration through block routing
Each service includes a block router that uses a routing table to control the flow of logic within the service. The routing table contains the order of execution of the blocks. The block router is responsible for calling the blocks as needed.

Figure 9: A block router within a service

When a block has an output to process using another block within the service, it issues a notification. The block router receives the notification, looks up the next block in the routing table, and calls the next block to continue processing.

If the block's output is external to the service, the block will contain the logic needed to handle the output itself. For example, if the block is to write to a database, the block will write directly to the database without notifying the block router.

The use of the block router and routing table enables each block to operate without any knowledge of other blocks within the service, which in turn enables the blocks to be run asynchronously and independently.

The service itself provides the cohesiveness needed for the blocks to work together to produce logical flows of data. Blocks and logic can be added, removed, and modified within a service without having to change the entire service.

The logic flow of a service is modifiable simply by altering the order of execution defined in the routing table.

Imagine that the service of Figure 9 receives a signal using Block A and the signal is passed to Block B. Block B processes the signal to determine whether a particular event has occurred. If the event has occurred, Block C is configured to send an email to a designated address. A simple routing table would be:

This means that when the block router receives a notification from Block A, it calls Block B, and when it receives a notification from Block B, it calls Block C.

Assume that a user wants to modify the service to also actuate a device when the event occurs. To accomplish this, the user adds Block D, which is configured to actuate the device based on the output of Block B. The user adds Block D to the service and modifies the routing table to be:

This modification means that the block router will now call both Blocks C and D when Block B notifies of an output and this is achieved without requiring any changes to Block B itself.
Service instantiation and lifecycle

A service may be instantiated multiple times. The use of different configuration information for different instantiations allows customization to occur at runtime using the same service class.

Figure 10: Different runtime configurations of a single service class

Services may be started and stopped independently of one another, and thus have a lifecycle of their own. Typically, each service will run in its own system process on the underlying hardware that the nio instance is running on.

Services may be standalone or configured to communicate with each other. By configuring services to provide output to other services, complex logic may be constructed within a nio instance using different services to handle different parts of the logic. Multiple instances of nio can be used to form nio systems, which are described in a later section.
the core: foundation of nio

The core forms the underlying structure of the nio Platform. The services run on the foundation provided by the core. They cannot be run without the core, just as blocks cannot be run without a service.

A nio instance does not know what services it will be running until it is launched, so the core is designed to be both flexible and consistent. The flexibility allows various modules of the core to be changed, such as communication and security modules, and allows different services and blocks to be run on any core. The consistency ensures that the core’s primary functionality for starting and configuring services remains the same and that services will always run in the same way.

**Figure 11:** The nio core in a nio instance

The core has four primary functions:
1. A service manager that
   - creates contexts for services
   - launches services
2. An external interface for instance and service management
3. Additional core management functionality
4. Determining instance interface implementations (e.g., security and communications)

**Service manager**

The core uses a service manager to identify the services assigned to the nio instance. The service manager creates a context for each service (described previously in Figure 10) simply as configuration information. In reality, the configuration information in Figure 10 is a service-specific context that contains both static, predefined configuration information and information that is dynamically generated at startup as illustrated in Figure 12. (Similarly, a service creates a context for each of the blocks assigned to the service and uses the context to configure the block when the service is launched.) The service is instantiated and then configured using the context.

**Figure 12:** Creation and application of a context to a service at runtime

A communications channel (typically an inter-process communication (IPC) channel) is maintained between the service manager and each service. This allows the service manager to monitor the service and send commands to the service, such as start and stop messages.
API for instance and service management
The core provides an API (typically via a REST interface) that enables external communications, such as queries and command messages. While much of the core's functionality is hidden from a user, varying levels of access may be provided via the API, including service and block configuration, and instance and service start/stop commands.

Additional management functionality - core components
In addition to managing the lifecycle of each service in their own system process, the core provides the ability to perform logic in the core process as well. This is done through the use of core components. These components are optional and run in the core process alongside the service manager and the API. Components may be used to augment the functionality of your nio instance and are not usable by blocks.

For example, a core component might expose an API to fetch logs or dynamically change the verbosity of logging of the nio instance. Another example would be a core component that connects to Google Cloud IoT Core and allows users to manage their nio services and blocks from the Google IoT Core Console.

Instance interfaces and implementations - modules
Every nio instance comes with interfaces, known as modules, that are available to be used by the core and by blocks.

There are currently six modules available for nio developers to use:
- Communication - Allows for the publishing and subscribing of data via string-based topic trees
- Persistence - Allows for the storage of data that can survive service and instance restarts or crashes
- Scheduler - Allows for scheduling repeatable or one-time tasks to take place some time in the future
- Security - Allows for the authorization and authentication of users
- Settings - Allows for the fetching of instance settings in blocks or modules
- Web - Allows for the exposure of web server ports and handlers

Module interfaces allow block developers to leverage the functionality of the core in their block code without having to worry about how the core performs the functionality or implement it themselves. For example, a block developer can persist some state using the persistence interface without worrying about whether that data is being saved in a database, a file on disk, or in a remote data store. The location and method of data persistence is a decision the core controls.

Every running nio core will have one and only one implementation of each module interface. This means that if you want to use different types of persistence implementations, you would need to run multiple nio instances. In addition, module implementations run in every service process. This is different than a core component which only runs in the core process.

Different nio binaries will often include different module implementations. For example, a lightweight hobbyist binary may rely on HTTP Basic Authentication for its security module implementation. However, an enterprise nio binary may use Active Directory or SAML for its security module. This allows systems integrators and architects to determine how a system should be secured, while letting block developers and service designers focus on the logic of the system.
communications: elegantly orchestrating data interactions

nio provides adaptable communications. The default communications layer, Pubkeeper, seamlessly bridges communications across protocols with a simple client. In addition, communications are configurable on a per-service basis. This means a single nio instance can communicate in multiple distinct ways with any number of devices.

nio cores do not communicate directly with one another; instead, communications between instances occur when a service running on an instance communicates with a service running on another instance.

While each nio instance has a default communications layer for its services, any individual service may be configured to use another communications method by using a separate block. A service’s communications are either with another service (service-to-service) or with a non-nio enabled device.

Service-to-service communications
Service-to-service communications may include services on the same instance or on different instances. A service does not care where another service resides (i.e., on the same or a different device) as long as the two services are configured to communicate. Services can communicate using a peer-to-peer model or a client-server model.

Service-to-service communication with Pubkeeper
Pubkeeper provides a publish-subscribe system and uses topic trees to organize information.

A Pubkeeper brew can be a lower-level protocol like UDP or TCP or can be a more advanced communication protocol like Apache Kafka or Google Cloud Pub/Sub. Pubkeeper uses “brews” that each represent a particular protocol. For example, an instance with brews for MQTT, HTTP, UDP, and Websockets can communicate using any of those protocols. Pubkeeper even enables web browsers to directly communicate with other Pubkeeper clients using Websockets, which allows browsers to be direct participants in nio systems.

In a Pubkeeper-enabled instance, the communications layer is provided by a Pubkeeper client that communicates with a Pubkeeper server and other Pubkeeper clients. Pubkeeper clients may be used by both nio instances and non-nio enabled devices. The Pubkeeper server keeps track of each client and the available topics, and manages the publish-subscribe system used by the clients. The

Figure 13: Service-to-service with Pubkeeper communication for Pubkeeper enabled services
Pubkeeper client makes the publish-subscribe model available to all services running on the instance.

Topics are structured in a hierarchical tree to allow a service to subscribe to multiple topics at once. For example, if services A, B, and C each want to publish temperature data, they can publish to topics `temperature.A`, `temperature.B`, and `temperature.C`, respectively. Another service can then subscribe to all temperature data by subscribing to `temperature.*` or to fewer than all of the topics. For example, if the service only needs B’s temperature data, it would subscribe to `temperature.B` to get only that data.

As long as services reside within the same Pubkeeper system, the actual location of each service is irrelevant. This means that a service can publish or subscribe to a topic without the need to know where the destination or originating service is located. This simplifies the distribution of services and logic to wherever they are needed in the system. Pubkeeper will automatically manage the inter-service communications.

Service-to-service communication with blocks

Although not commonly used due to Pubkeeper, services can be configured to communicate with one another using blocks that implement particular peer-to-peer or client-server models. For example, a service may use an HTTP publisher block to send data directly to another service that receives the data using an HTTP handler block.

Non-nio communications

If a non-nio device is running a Pubkeeper client, a service may communicate with the device as with any nio instance running Pubkeeper.

![Figure 15: Service to non-nio device communication using Pubkeeper](image)

If the non-nio device is not running a Pubkeeper client, the service may communicate directly with the device using blocks configured for input and/or output. This enables a service to communicate in any way necessary: HTTP or HTTPS access to a server, direct access to microcontrollers using the microcontroller’s I/O pins, and direct connections to any other digital signal source/sink, including analog/digital converters.

![Figure 16: Service to non-nio device communication without Pubkeeper](image)

COMMUNICATION SECURITY: END-TO-END ENCRYPTION

Pubkeeper facilitates end-to-end encryption of communication channels. When a match between a publisher and a subscriber is made, Pubkeeper informs both parties about which protocols should be used and manages the shared encryption keys. Because the encryption is end-to-end, only the originating and destination instances are able to access the unencrypted data. Any intermediate nodes are unable to decrypt the data because they are not provided the shared keys.
systems: putting it all together

We previously defined a nio system as a connected group of nio instances and other entities, if any. There is no requirement to run nio on every device in a system, but doing so provides many benefits, such as ease of development and deployment, secure communications, and real-time monitoring.

Each nio system has one and only one Pubkeeper server that orchestrates the system’s communications. It is important to keep this in mind as you determine where to delineate your systems. Instances that need to communicate with each other directly should be added to the same system.

Of course, not every instance in a system needs to talk to every other instance. Systems come in all shapes and sizes. Solution architects can design peer-to-peer, client-server, or hybrid systems with nio.

This paper dives into the technical details of nio, but the most interesting part of nio will always be what you decide to implement with it. We provide this foundational knowledge so you may confidently build your own amazing solutions with nio. We have spent years imagining and building the most adaptable, modular, and progressive platform architecture possible, so you can do the same with your use cases.

In closing, our goal is to empower solution architects in three steps:

1. Iterate the use case and problem solving without technical constraint
2. Easily create system logic using powerful tools
3. Distribute logic optimally throughout the system

We will constantly chase that ideal with nio. We can’t wait to see what you build.

Questions or want to talk to a niolabs representative? Contact us at info@n.io
For an even deeper dive into the architecture, check out our patents here.

want to see nio in action?
request a private demo

digital transformation starts with a plan
start building today

want special inside access?
become a nio influencer
DISTRIBUTED LOGIC: VINEYARD USE CASE

A truly distributed system implies that logic and computation are performed at all layers of the system. Simply connecting a device to the internet and shipping data up to the cloud for processing is not efficient or feasible for every use case or at trillions scale.

Running nio throughout your distributed system means you can write logic and deploy it anywhere you want, not just the cloud. Logic embodied in blocks may be distributed across different services, and those services may be distributed across different instances that are all connected by Pubkeeper.

For example, nio is currently used as a comprehensive solution for vineyard irrigation control and automation at Deep Sky Vineyard. By installing nio at each level of the system (edge, gateway, and cloud), we distribute logic to create an efficient, robust solution that follows a loop learning model. Here’s how it works:

**Measure, Understand, & Act (Edge)**

With thousands of rows of grapes being monitored across multiple variables, the amount of data produced on the vineyard is immense. Rather than shipping all the raw sensor data to the cloud to make every watering decision, this solution applies edge logic and actuation at the field-level device in each watering zone. This allows each zone to request exactly how much water it needs from the centralized irrigation controller based on the current plant/soil conditions and its watering algorithm. Having edge intelligence also means that each zone can take immediate actions independent of higher-level devices—like closing a valve in the event of a downline irrigation break, which can be sensed by flow and pressure data.

**Model, Understand (Gateway)**

Each field-level request for water is transmitted to a higher-level instance, in this case located in the pump house. This intermediate node not only has the context of each watering zone, but also an on-site weather station. It prioritizes the water requests to optimize the watering schedule and make sure the zone most in need receives water first. It can overrule the water requests based on weather events like rain. It can also share understanding gained from its holistic perspective down to the edge nodes to adjust their watering algorithms.

**Model, Understand (Cloud)**

In this solution, a cloud instance can be employed for weather forecasting via cloud APIs and application of machine learning to watering algorithms. Models may be trained in the cloud and then deployed to the edge. This architecture is not only efficient, but robust. Routine decisions are made that locally limit networking costs and the impact of loss of connectivity. High-value, computationally intensive functions are executed in the elastic cloud.
architecting solutions with nio

idea

problem solve
without constraint

nio

create
system logic

distribute
logic anywhere

devices