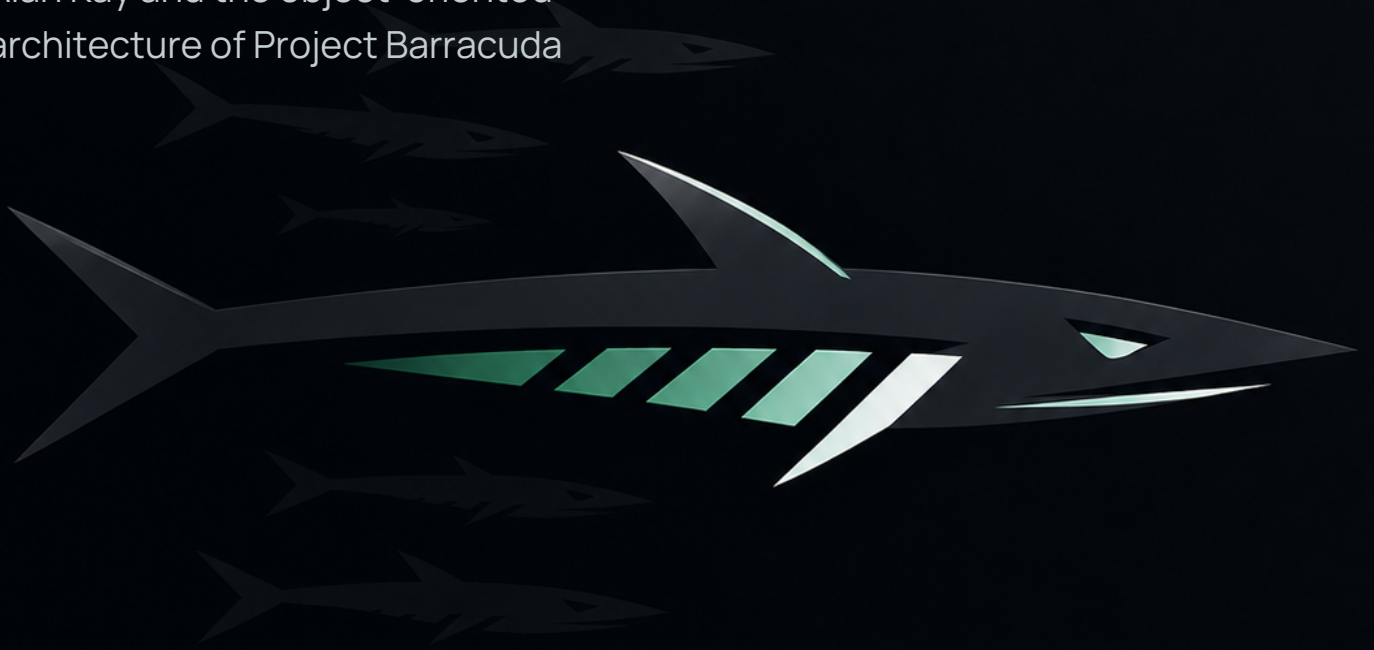


The Big Idea Is Still Messaging

Alan Kay and the object-oriented
architecture of Project Barracuda



Project Barracuda

Litmus Automation has been quietly at work on Project Barracuda, a new application built on NATS, Go, and OPC Unified Architecture (OPC UA). A few months into the project I happened to read Steve Jobs by Walter Isaacson, and I came back to Alan Kay's original idea of objects as computers on a network. To Kay, objects are independent parts that own their state and interact through messages. Barracuda is our attempt to take that idea seriously for industrial systems. On the factory floor, that means a new participant, human or AI, is just another sender or subscriber.

About the project

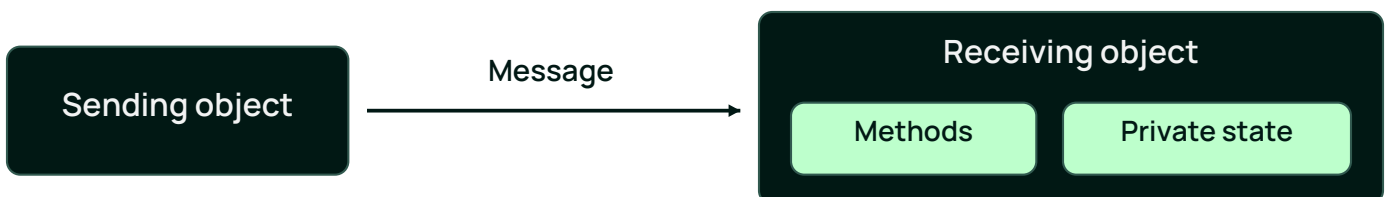
Barracuda is an industrial data server with the shape of a common distributed industrial software component - an OPC server with natively-integrated device drivers built for multi-tasking operating systems. Inside, however, its design keeps the object interface separate from the message layer, and yet treats both as first-class interfaces.

The server is written in Go and compiles to a single binary for mixed CPU architectures and operating systems found across industrial settings. Once running, its components (drivers, servers, other modules) are loosely coupled across an internal NATS bus, but they share a single contract precise enough to act as an OPC UA server for process control. That contract is implemented as the object's interface, and the interface stays independent of how messages reach it. This means that for data acquisition, users can treat Barracuda's own internal message bus as a rich external event source with query, publish and subscribe using native NATS or another messaging protocol. Clients that need OPC UA's contracted obligations and wire protocol use a Barracuda OPC UA server.

Kay's idea

In a 1998 message to the Squeak/Smalltalk community, Kay wrote, "The big idea is 'messaging'." He argued that building systems that scale depends more on how modules communicate than on what is inside them. In 2003, he compared objects to biological cells or computers on a network, and defined object-oriented programming this way:

"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things." (Kay, 1998, 2003)



A caller asks an object to do something, and the object owns how that request is handled. We can understand the interaction without knowing every detail of the implementation. That is a useful way to think about drivers, servers, system models, and the software that manages them.

Objects that describe themselves

Industry gets above-average benefit from object modeling and object-oriented programming. A plant has many systems to integrate, each with its own behavior. Its machines are often complex, critical, remote, or isolated, which makes digital twins more useful, whether for process execution or simulation. And industry has OPC UA Parts 3 and 5, which compose into a mature, standard way to define reusable objects and their types, data, operations, and relationships.

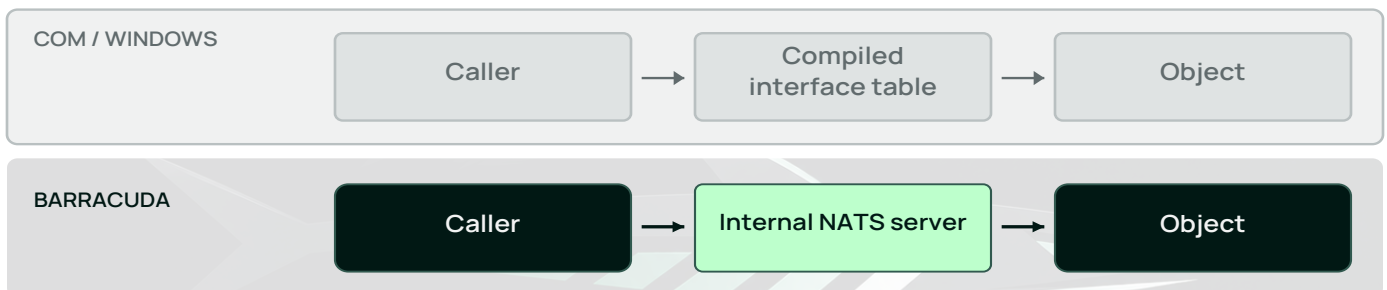
Steve Jobs met Kay at the famous and consequential Xerox PARC tour, and this was the start of a close working relationship for decades. Jobs shared his enthusiasm for objects. At NeXT, OPENSTEP put that enthusiasm to work through reusable application frameworks. WebObjects carried the approach into web applications, connecting business objects to database records, and developers worked through interfaces while objects owned their implementation. An object-oriented interface defines a contract, and publishing that contract for other software to inspect makes the interface self-describing.

In a distributed industrial system, a caller has to discover a machine's type, data, and operations. Without a self-describing interface, that discovery means complex client-side logic, the kind NeXT engineered out of its own system.

Industrial Interfaces through Microsoft COM

Microsoft introduced the Component Object Model (COM) in 1993, while NeXT was building its object frameworks. COM defined an object as a binary interface any language could call, and DCOM carried that interface across the network. OPC Classic was built on COM and DCOM. The OPC Foundation's members succeeded in defining and implementing distributed object-oriented client-server software standards on networks of Windows-based PCs - for process control and SCADA no less!

But COM paid for binary compatibility with most of Kay's messaging. A call went through a compiled interface table to one object on one platform, so what an object was and how you reached it became one thing. Components were easy to reuse and hard to deploy, and OPC Classic stayed tied to Windows and DCOM. OPC UA loosened that tie with a platform-independent wire protocol and its own information model. Barracuda keeps the object contract and the message layer apart, so neither the platform nor the protocol decides what an object is.



OPC UA gives industrial objects meaning

In OPC UA, clients browse Objects, Variables, Methods, and the references between them. ObjectTypes and subtypes form a data structure that makes traditionally difficult and time-consuming queries easy and cheap, and companion specifications extend base definitions with more industrial vocabulary. As of 2026, the OPC Foundation's working groups, partner organizations, and outside organizations have created over 430 definitions of physical equipment and software systems using OPC UA's type system. We believe there is significant value in pre-created OPC UA NodeSets for functional asset and system models, and the definition language of the NodeSets - UA Part 5 - provides a flexible and general-purpose class-based functional modeling language. Part 5 provides both contextual features and functional services for its models. We can describe services with inputs and outputs that intend to be invoked by external clients. It allows us to describe Barracuda's own components and object-oriented interfaces using the same definition language we use to describe data sets and external systems, and this makes what normally are hidden, implicit relationships like that of a server interface and driver and data set and target devices natively explicit. With all relationships natively explicit, discovery and derivation is fast and cheap for developers and users with traditional tools, analytics or AI.

So, it is important to us that Barracuda have deep support for custom type definitions using OPC UA Part 5 as well as NodeSet2 files regardless of whether users intend to utilize our OPC UA server. Barracuda's shared namespace is thus represented as an OPC UA information model, and a Barracuda UA server exposes a view of it.

OPC UA

Model

Go

Behavior

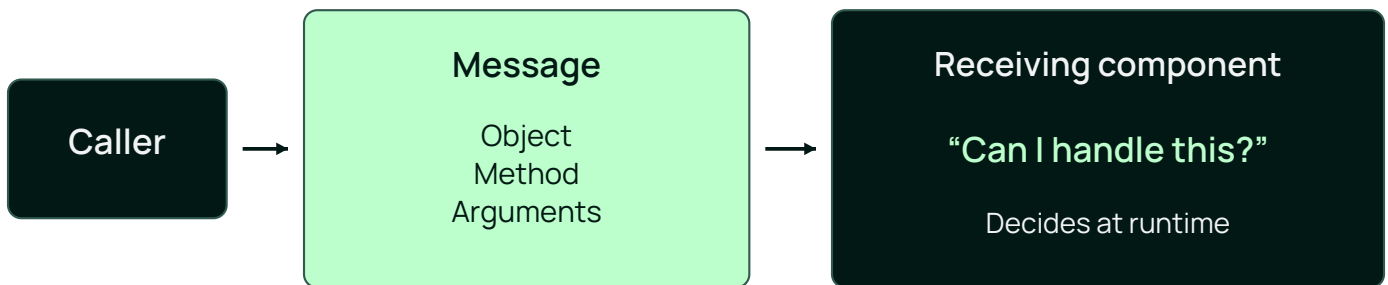
NATS

Messages

From model to executable behavior

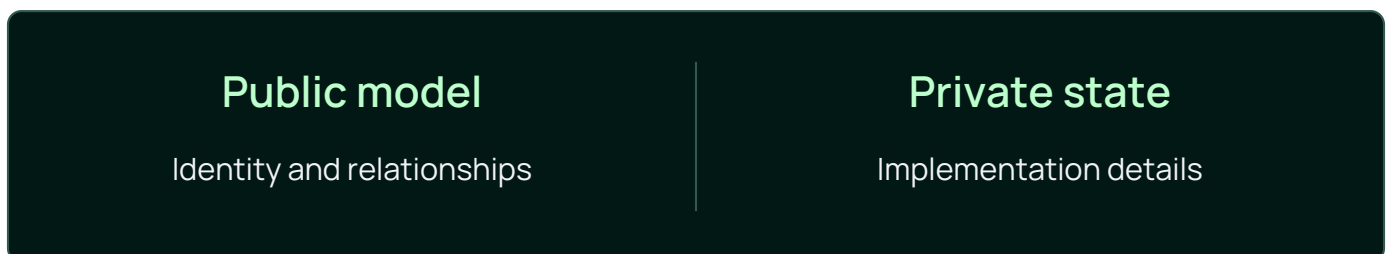
Go is where an object's behavior lives. In the job example in the next section, a software component validates the order, translates it into controller commands, and decides what a failure means. The OPC UA model only says the operation exists, and NATS only delivers the request. Go adds its narrow slice. Methods attach to any type you define, and an interface lists what a caller needs. Any type with those methods satisfies it, so drivers, servers, and adapters can be swapped without changing the caller.

Go stops short of Kay's "extreme late-binding of all things" because method sets are fixed at compile time. Barracuda recovers some of it at the message layer. A call names its object and method by NodeId as data, and the receiving component decides at runtime whether it handles it.



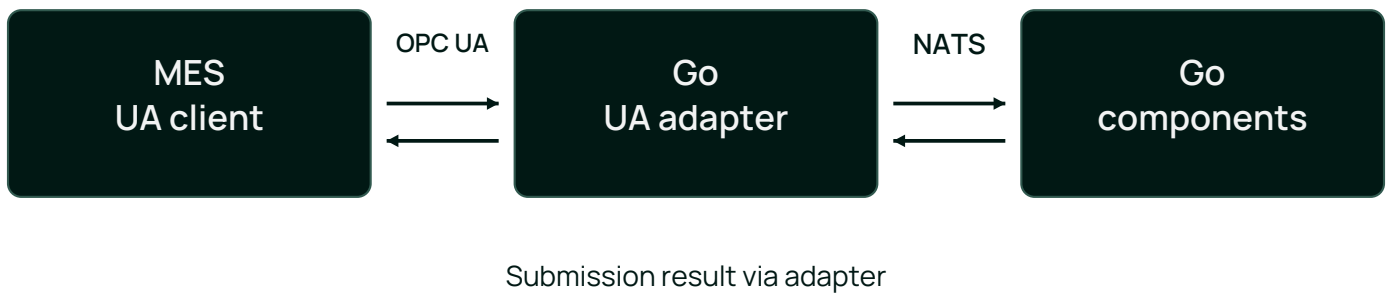
On the model side, richer objects are optional: an object type can carry Variables, Methods, and subtypes using OPC UA's type system (Parts 3 and 5). Type definitions can be created and changed over NATS, or imported from standard NodeSet2 files such as companion specifications.

Components declare their identities, and arriving device values can create device and variable nodes, so data is available as plain scalar values with no modeling. Mapping rules may then assign more specific UA types. The shared namespace lives in a NATS JetStream key-value bucket. It holds public model state whereas each implementation keeps its runtime state private.



An object operation becomes a message

Consider job J-104: produce 500 brackets on machine M-7. An MES application uses OPC UA to call the StoreAndStart Method on the machine's job receiver, passing a typed job order with its ID and production parameters. ISA-95 Job Control and Machinery Job Management provide the vocabulary.



A Go UA adapter packages the object identity, Method identity, and arguments into a typed NATS request. Internal components receive it, validate the job, and translate it into the controller's commands. The reply travels back through the adapter as the OPC UA Method result. Direct NATS callers use the same request and response contract.

One data foundation

On a factory floor, Kay's independent parts, computers, are machines, and the participants around them keep multiplying: operators, control systems, analytics, and now AI agents. If each one integrates with a machine's own interface, every new participant means new integration work. If they all address the same objects through messages, a new participant is just another sender or subscriber, and the objects stay the same. That shared understanding is a critical part of the factory of the future where people, systems, and AI agents coordinate their work through common knowledge of the equipment, data, and available operations. Because we kept the message layer separate from the object interface just as Kay argued, each participant reaches the same objects in the way that suits it: OPC UA where its obligations matter, plain messages where they do not.

Research we can build on

The research we've collected includes academic and industry work on automatic OPC UA configuration, automated information-model generation, and semantic OPC UA skills. These papers explore how systems can acquire a model and expose reusable capabilities. Barracuda brings those questions into our own combination of an OPC UA model, Go behavior, and NATS messaging.

More to come. Until then, LET'S GET MODELING!

Selected sources

Alan Kay

- Kay, A. Squeak-dev message (10 October 1998).
- Kay, A. Correspondence with Stefan Ram (23 July 2003).

Apple, NeXT, and Steve Jobs

- Isaacson, W. Steve Jobs. Simon & Schuster, 2011.
- Jobs, S. Smithsonian oral history (20 April 1995)
- NeXT, OpenStep and the return of Steve Jobs to Apple
- Apple. Cocoa Fundamentals Guide: What Is Cocoa?
- Apple. How WebObjects Works; WebObjects 5 announcement (21 May 2001).

OPC Foundation

- OPC Foundation. Part 5, Information Model, v1.05.06; Part 3, Address Space Model, v1.05.06; Part 4, Services, v1.05.07.
- OPC Foundation: OPC Classic; history and OPC UA 1.0 release; companion specification count (20 April 2026).
- OPC Foundation: ISA-95, Part 4: Job Control, StoreAndStart and job responses, v2.0; Machinery, Part 3: Job Management, v1.0.1
- OPC Foundation, OPC 10000-14, PubSub, v1.05.06.
- OPC Foundation. OPC UA Part 3, Standard ReferenceTypes, v1.05.06.

Academic research

- Gutierrez-Guerrero, J. M., and Holgado-Terriza, J. A. (2019). Automatic Configuration of OPC UA for Industrial Internet of Things Environments. Electronics, 8(6), 600.
- Busboom, A. (2024). Automated generation of OPC UA information models - A review and outlook. Journal of Industrial Information Integration, 39, 100602.
- Profanter, S., Perzylo, A., Rickert, M., and Knoll, A. (2021). A Generic Plug & Produce System Composed of Semantic OPC UA Skills. IEEE Open Journal of the Industrial Electronics Society, 2, 128-141.

Go and NATS

- Go FAQ: Is Go an object-oriented language?
- Core NATS, request-reply, JetStream acknowledgment and redelivery, JetStream consumers, and key-value storage.
- NATS Go client v1.52.0: micro/service.go, micro/request.go