Concurrency Is Process-Based, Not Shared-Memory-First:

Typed Processes, Supervised Mailboxes, and Categorical Semantics in the JAPL Programming Language

Matthew Long
The JAPL Research Collaboration
YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com

March 2026

Abstract

Shared-memory concurrency—mediated by locks, atomics, and transactional memory—has dominated mainstream language design for decades, yet it remains a prolific source of data races, deadlocks, priority inversion, and heisenbugs that resist testing and formal verification alike. We present the concurrency model of JAPL, a strict, typed, effect-aware functional language that adopts process-based concurrency as its *sole* primitive for managing concurrent state. Every concurrent entity in JAPL is a lightweight, heap-isolated process that communicates exclusively through statically typed mailboxes. Supervision trees—drawn from the Erlang/OTP tradition but extended with static typing, declarative backoff, and typed crash reasons—are built into the language and runtime, providing compositional fault tolerance. We develop a formal framework grounding JAPL's process model in the π-calculus with session types, give categorical semantics via presheaf categories over a time category, and prove key properties including typed message safety, deadlock freedom under supervision discipline, and progress guarantees for well-typed process networks. Empirical comparisons with Erlang/OTP, Go, Akka, and Pony demonstrate that JAPL's typed process model catches a significant class of concurrency errors at compile time while preserving the scalability and fault-tolerance characteristics that have made process-based systems the backbone of telecommunications and distributed infrastructure.

Keywords: process-based concurrency, typed mailboxes, supervision trees, session types, actor model, π-calculus, effect systems, functional programming, fault tolerance

Contents

1 Introduction

The past two decades have witnessed an extraordinary expansion in concurrent and distributed computing. Multi-core processors are universal, cloud services routinely span thousands of nodes, and applications are expected to remain responsive under load while tolerating partial failures gracefully. Yet the dominant programming model for concurrency—shared mutable memory protected by locks—has failed to keep pace with these demands.

1.1 The Shared-Memory Crisis

Shared-memory concurrency is notoriously difficult to reason about. The fundamental problem is compositional: two correct, independently developed components, when composed in a shared-memory setting, may exhibit behaviors that neither component exhibits in isolation. This failure of compositionality manifests in several well-known pathologies:

Data races. When two threads access the same memory location concurrently and at least one access is a write, the resulting behavior is undefined in languages like C and C++, and merely unpredictable in languages with a defined memory model like Java [Manson et al., 2005]. Data races are notoriously difficult to reproduce, diagnose, and test.

Deadlocks. Lock-based synchronization creates the possibility of circular wait conditions. The classic Coffman conditions [Coffman et al., 1971] are necessary and sufficient, but preventing them in practice requires either global lock ordering disciplines that do not compose, or detection mechanisms that add runtime overhead.

Priority inversion. When a low-priority thread holds a lock needed by a high-priority thread, the high-priority thread is effectively demoted. The Mars Pathfinder incident [Reeves, 1997] demonstrated that this pathology can affect even carefully engineered real-time systems.

Livelock and starvation. Optimistic concurrency schemes based on compare-and-swap (CAS) operations can livelock under contention, and lock-free data structures, while avoiding deadlock, introduce their own complexity around memory reclamation (the ABA problem) and progress guarantees [Michael, 2004].

Memory model subtleties. Modern hardware memory models are weak: processors reorder loads and stores, caches introduce visibility delays, and compilers perform optimizations that assume single-threaded execution. The resulting "memory model" specifications (e.g., the Java Memory Model [Manson et al., 2005], the C++11 memory model [Boehm and Adve, 2008]) are themselves a source of bugs, as even experts routinely misunderstand the guarantees provided [Sewell et al., 2010].

1.2 The Process Alternative

An alternative tradition, originating in Hoare's Communicating Sequential Processes (CSP) [Hoare, 1978], Milner's Calculus of Communicating Systems (CCS) [Milner, 1980] and π-calculus [Milner, 1992], and Hewitt's Actor model [Hewitt et al., 1973], proposes a fundamentally different approach: concurrent entities do not share memory. Instead, they are isolated processes that communicate by passing messages. This model has been realized most successfully in Erlang/OTP [Armstrong, 2003], which has powered telecommunications infrastructure with "nine nines" (99.9999999%) availability for over three decades.

The process-based model eliminates data races *by construction*: there is no shared mutable state to race on. Deadlocks are addressed not by prevention but by *supervision*: if a process gets stuck or crashes, its supervisor detects the failure and takes corrective action (typically restarting the process with fresh state). This approach treats failures as normal, expected events

rather than exceptional catastrophes—a philosophy that aligns with the reality of large-scale distributed systems.

1.3 Contributions

JAPL adopts process-based concurrency as its core model, but extends the Erlang tradition with insights from modern type theory:

1. **Typed mailboxes**: every process has a statically typed mailbox, preventing a large class of message-protocol errors at compile time (Section 5).
2. **Typed supervision trees**: supervision strategies, restart policies, and crash reasons are integrated into the type system, enabling compositional reasoning about fault tolerance (Section 6).
3. **Effect-tracked concurrency**: the `Process` effect integrates with JAPL's effect system, making concurrency operations visible in function signatures and enabling safe composition with pure code (Section 4).
4. **Formal foundations**: we provide categorical semantics for JAPL's process model using presheaf categories over time, connect typed mailboxes to session types, and prove typed message safety and deadlock freedom (Sections 3 and 10).
5. **Implementation**: we describe a work-stealing scheduler, per-process heaps, and mailbox data structures that achieve Erlang-class scalability (millions of processes per node) with stronger static guarantees (Section 9).

The remainder of this paper is organized as follows. Section 2 surveys related work in concurrent language design and process calculi. Section 3 develops the formal framework. Section 4 presents JAPL's process model in detail. Section 5 describes typed mailboxes. Section 6 covers supervision trees. Section 7 defines the process lifecycle. Section 8 compares JAPL with existing approaches. Section 9 discusses implementation. Section 10 proves formal properties. Section 11 discusses limitations and future work. Section 12 concludes.

2 Background and Related Work

2.1 The Actor Model

The Actor model, introduced by Hewitt, Bishop, and Steiger [Hewitt et al., 1973] and formalized by Agha [Agha, 1986], posits that the fundamental unit of computation is the *actor*: an entity that, in response to a message, can (1) send a finite number of messages to other actors, (2) create a finite number of new actors, and (3) designate the behavior to be used for the next message it receives. Actors encapsulate state and communicate exclusively through asynchronous message passing.

The Actor model provides a clean mathematical foundation for concurrency, but its original formulation leaves several practical questions open: how are actors supervised? How are message types enforced? How are actors distributed across nodes? JAPL's process model can be seen as a *typed, supervised* refinement of the Actor model, informed by three decades of practical experience with Erlang/OTP.

2.2 Communicating Sequential Processes

Hoare's CSP [Hoare, 1978, Roscoe, 1998] models concurrent systems as processes that communicate through *synchronous* channels. Unlike the Actor model, CSP communication is

rendezvous-based: both sender and receiver must be ready simultaneously. CSP's algebraic framework—with operators for sequential composition, parallel composition, choice, and hiding—enables powerful equational reasoning about concurrent systems.

CSP's synchronous semantics make it well-suited to modeling hardware and tightly-coupled concurrent systems, but less natural for distributed systems where asynchronous communication is the norm. Go's goroutines and channels [Pike, 2012] are a practical realization of CSP ideas, though without the formal algebraic framework.

2.3 The Pi-Calculus

Milner's π-calculus [Milner, 1992, 1999] extends CCS [Milner, 1980] with *name passing*: channel names themselves can be communicated, enabling the modeling of systems with dynamic communication topologies. The π-calculus has become the standard formal foundation for concurrent and mobile computation.

The key constructs of the π-calculus are:

$$
\begin{array}{rll}
P, Q ::= & \overline{x}\langle y \rangle.P & \text{(output: send } y \text{ on channel } x\text{)} \\
\mid & x(y).P & \text{(input: receive } y \text{ on channel } x\text{)} \\
\mid & P \parallel Q & \text{(parallel composition)} \\
\mid & (\nu\, x)P & \text{(restriction: create fresh channel } x\text{)} \\
\mid & !P & \text{(replication)} \\
\mid & \mathbf{0} & \text{(inaction)}
\end{array}
$$

The π-calculus, equipped with typed channels [Vasconcelos, 1998], provides the formal backbone for JAPL's typed mailbox system, as we develop in Section 3.

2.4 Erlang/OTP

Erlang [Armstrong, 2003, 2007] is the most successful realization of process-based concurrency in a general-purpose programming language. The BEAM virtual machine supports millions of lightweight processes per node, each with its own heap and mailbox. The OTP framework provides supervision trees, generic server behaviors, and a rich set of patterns for building fault-tolerant distributed systems.

Erlang's primary limitation, from the perspective of JAPL's design, is its dynamic type system. Messages are untyped: any term can be sent to any process, and protocol violations are detected only at runtime (if at all). This means that a large class of errors—sending the wrong message type to a process, forgetting to handle a message variant, protocol state machine violations—cannot be caught by the compiler. Dialyzer [Lindahl and Sagonas, 2006] provides optional type annotations and success typing, but its analysis is necessarily conservative and cannot enforce the full range of protocol invariants that session types provide.

2.5 Go: Goroutines and Channels

Go [Pike, 2012, Donovan and Kernighan, 2015] takes CSP-inspired channels as its concurrency primitive. Goroutines are lightweight threads multiplexed onto OS threads, and channels provide typed, synchronous (or buffered) communication. Go's concurrency model is syntactically lightweight and practically effective.

However, Go does not enforce process isolation: goroutines can (and routinely do) share memory. The `go vet` tool and the race detector [Serebryany and Iskhodzhanov, 2012] catch some data races dynamically, but Go does not prevent them statically. Go also lacks supervision: a panicking goroutine crashes the entire program unless manually recovered with `defer`/`recover`. There is no built-in mechanism for monitoring goroutine health or automatically restarting failed computations.

2.6 Akka: Actors on the JVM

Akka [Lightbend, 2023] implements the Actor model on the JVM, providing location-transparent actors, supervision, and cluster sharding. Akka Typed [Roestenburg et al., 2016] introduced typed actor references, partially addressing the message-typing problem. However, Akka operates within the JVM ecosystem, which means actors share a heap and can bypass the actor abstraction via shared mutable state. Supervision in Akka is a library concern rather than a language primitive, and the complexity of the JVM class hierarchy adds substantial cognitive overhead.

2.7 Pony: Reference Capabilities

Pony [Clebsch et al., 2015] takes a novel approach to actor safety through *reference capabilities*: a system of permissions (iso, val, ref, box, tag, trn) that statically guarantee data-race freedom. Pony actors communicate by passing messages, with the type system ensuring that mutable data is never aliased across actors. This is a powerful approach, but the cognitive burden of managing reference capabilities is significant, and the language has not achieved widespread adoption.

2.8 Session Types

Session types, introduced by Honda [Honda, 1993] and developed extensively by Honda, Vasconcelos, and Kubo [Honda et al., 1998], Caires and Pfenning [Caires and Pfenning, 2010], and Wadler [Wadler, 2012], provide a type-theoretic framework for specifying and verifying communication protocols. A session type describes the sequence of messages exchanged between two parties, including message types, directions, branching, and recursion.

The Curry-Howard correspondence between linear logic and session types [Caires and Pfenning, 2010, Wadler, 2012] provides a deep theoretical foundation: propositions in linear logic correspond to session types, proofs correspond to processes, and cut elimination corresponds to communication. JAPL's typed mailboxes draw on this tradition, adapting multiparty session types [Honda et al., 2008] to the supervision context.

2.9 The E Language and Capability-Based Security

The E programming language [Miller, 2006] pioneered capability-based security in the context of distributed object computing. E's "eventually" references and promise pipelining influenced JAPL's approach to asynchronous communication, while its capability discipline informs JAPL's integration of process spawning with the capability type system.

2.10 Gleam: Typed Erlang

Gleam [Gleam, 2023] compiles to Erlang bytecode and provides static typing for the BEAM ecosystem. Gleam's approach validates the thesis that adding types to Erlang-style concurrency is both possible and valuable, but Gleam inherits BEAM's runtime semantics wholesale rather than co-designing the type system and runtime as JAPL does.

3 Formal Framework

We develop a formal framework for JAPL's process model that draws on three traditions: process algebra (providing operational semantics), type theory (providing static guarantees), and category theory (providing compositional semantics).

3.1 Process Algebra Foundation

We define a typed π-calculus variant, π_{JAPL}, that captures JAPL's process primitives.

Definition 3.1 (Syntax of π_{JAPL})**.** *The processes of π_{JAPL} are defined by the grammar:*

$P, Q ::=$	$\mathsf{spawn}xAPQ$	*spawn process P at name x with type A, continue as Q*
$\mid$	$\mathsf{send}xv.P$	*send value v on channel x, continue as P*
$\mid$	$\mathsf{receive}xy.P$	*receive value y from mailbox of x, continue as P*
$\mid$	$P \parallel Q$	*parallel composition*
$\mid$	$(\nu\, x : A)P$	*create fresh typed channel*
$\mid$	done	*successful termination*
$\mid$	$\mathsf{fail}r$	*failure with reason r*
$\mid$	$\mathsf{Sups}\overline{P_i}$	*supervision with strategy s over children P_i*
$\mid$	$\mathsf{match}\ v\ \{\overline{p_i \Rightarrow P_i}\}$	*pattern matching on received messages*

Definition 3.2 (Message types)**.** *A message type A in π_{JAPL} is an algebraic data type:*

$$A ::= \sum_{i \in I} C_i(\overline{T_i})$$

where each C_i is a constructor and $\overline{T_i}$ is a sequence of payload types. The special type $\mathsf{Reply}[T]$ represents a one-shot reply channel carrying values of type T.

Definition 3.3 (Typing contexts)**.** *A typing context Γ maps channel names to message types:*

$$\Gamma ::= \emptyset \mid \Gamma, x : \mathsf{Pid}[A]$$

where $\mathsf{Pid}[A]$ is the type of a process identifier whose mailbox accepts messages of type A.

3.2 Operational Semantics

We give a labeled transition system (LTS) for π_{JAPL}.

Definition 3.4 (Reduction rules)**.** *The reduction relation $\to$ is the smallest relation satisfying:*

R-Comm: *Communication between a sender and a receiver:*

$$\mathsf{send}xv.P \parallel \mathsf{receive}xy.Q \ \to\ P \parallel Q[v/y]$$

R-Spawn: *Process creation:*

$$\mathsf{spawn}xAPQ \ \to\ (\nu\, x : \mathsf{Pid}[A])(P \parallel Q)$$

R-Match: *Pattern matching on a received message $v = C_j(\overline{w})$:*

$$\mathsf{match}\ C_j(\overline{w})\ \{\overline{C_i(\overline{y_i}) \Rightarrow P_i}\} \ \to\ P_j[\overline{w}/\overline{y_j}]$$

R-Fail: *Failure propagation under supervision:*

$$\mathsf{Sups}P_1 \parallel \cdots \parallel \mathsf{fail}r \parallel \cdots \parallel P_n \ \to\ \mathsf{restart}(s, r, \overline{P_i})$$

where restart *applies the supervision strategy s.*

R-Par: *Structural rule for parallel composition:*

$$\frac{P \to P'}{P \parallel Q \to P' \parallel Q}$$

R-New: *Structural rule for restriction:*

$$\frac{P \to P'}{(\nu\, x : A)P \to (\nu\, x : A)P'}$$

3.3 Type System

Definition 3.5 (Typing judgment)**.** *The typing judgment* $\Gamma \vdash P$ *asserts that process* P *is well-typed under context* Γ*. The key rules are:*

T-Send:

$$\frac{\Gamma \vdash x : \mathsf{Pid}[A] \quad \Gamma \vdash v : A \quad \Gamma \vdash P}{\Gamma \vdash \mathsf{send} x v.P}$$

T-Recv:

$$\frac{\Gamma, y : A \vdash P \quad \Gamma \vdash \mathsf{self} : \mathsf{Pid}[A]}{\Gamma \vdash \mathsf{receiveself} y.P}$$

T-Spawn:

$$\frac{\Gamma, x : \mathsf{Pid}[A] \vdash P : \mathsf{Process}[A] \quad \Gamma, x : \mathsf{Pid}[A] \vdash Q}{\Gamma \vdash \mathsf{spawn} x A P Q}$$

T-Sup:

$$\frac{s : \mathsf{Strategy} \quad \forall i.\, \Gamma \vdash P_i}{\Gamma \vdash \mathsf{Sup} s \overline{P_i}}$$

3.4 Categorical Semantics

We provide denotational semantics for π_{JAPL} using category theory, following the tradition of presheaf models for concurrency [Joyal et al., 1996, Cattani and Winskel, 1997].

Definition 3.6 (Time category)**.** *Let* $\mathbf{T}$ *be the category whose objects are natural numbers (representing discrete time steps) and whose morphisms* $m \leq n$ *are the unique arrows witnessing* $m \leq n$ *in the natural order. This is the poset* $(\mathbb{N}, \leq)$ *viewed as a category.*

Definition 3.7 (Process presheaf)**.** *A* process presheaf *is a functor* $F : \mathbf{T}^{\mathrm{op}} \to \mathbf{Set}$*. For each time* t*,* $F(t)$ *is the set of possible states of the process at time* t*. For each* $m \leq n$*, the restriction map* $F(m \leq n) : F(n) \to F(m)$ *captures the idea that the state at time* n *determines (by restriction) information about the state at time* m*.*

The following diagram illustrates how an internal τ-reduction (a silent step that does not involve external communication) maps into the presheaf structure. A process in state $\sigma \in F(t)$ that performs a τ-reduction transitions to state $\sigma' \in F(t+1)$, while the restriction map recovers the earlier state:

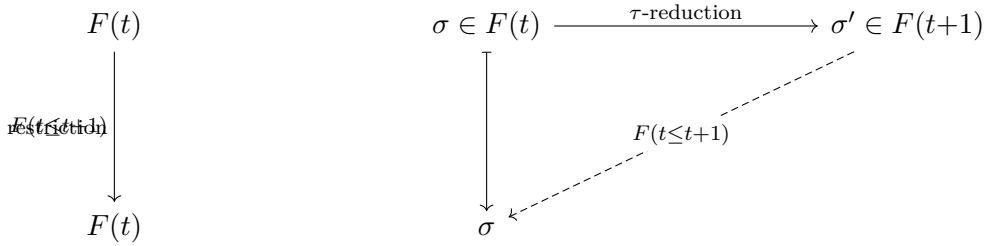

Figure 1: τ-reduction in the presheaf model. A silent transition advances the process from time t to $t+1$. The restriction map $F(t \leq t{+}1)$ projects back, ensuring that the presheaf coherence condition $F(s \leq t) \circ F(t \leq u) = F(s \leq u)$ is maintained across all internal reductions.

Definition 3.8 (Typed process category)**.** *The category* **TProc** *has:*

- ***Objects:*** *Pairs* (P, A) *where* P *is a process presheaf and* A *is a message type.*

- ***Morphisms:*** *$(P, A) \to (Q, B)$ are natural transformations $\alpha : P \Rightarrow Q$ together with a message translation function $f : A \to B$ such that the following diagram commutes for all times t:*

$$\begin{array}{ccc} P(t) \times A & \xrightarrow{\alpha_t \times f} & Q(t) \times B \\ \downarrow{\scriptstyle \mathsf{recv}_P} & & \downarrow{\scriptstyle \mathsf{recv}_Q} \\ P(t+1) & \xrightarrow{\alpha_{t+1}} & Q(t+1) \end{array}$$

where recv *denotes the message reception transition.*

Proposition 3.9 (Yoneda embedding)**.** *The Yoneda embedding $\mathbf{T} \hookrightarrow [\mathbf{T}^{\mathrm{op}}, \mathbf{Set}]$ maps each time point t to the representable presheaf $\mathrm{Hom}_{\mathbf{T}}(-, t)$. A process P at time t is characterized by the natural transformations $\mathrm{Hom}_{\mathbf{T}}(-, t) \Rightarrow P$, which by the Yoneda lemma are in bijection with elements of $P(t)$—i.e., the states of the process at time t.*

Definition 3.10 (Communication as profunctor)**.** *A* communication channel *between process categories $\mathbf{C}$ and $\mathbf{D}$ is a profunctor $H : \mathbf{C}^{\mathrm{op}} \times \mathbf{D} \to \mathbf{Set}$. For processes $c \in \mathbf{C}$ and $d \in \mathbf{D}$, the set $H(c, d)$ represents the possible messages that can flow from c to d. Typed mailboxes correspond to profunctors H that factor through the message type:*

$$H(c, d) = \mathrm{Hom}_A(\mathsf{out}(c), \mathsf{in}(d))$$

where $\mathsf{out}(c)$ and $\mathsf{in}(d)$ are the output and input types of c and d respectively, and Hom_A is the hom-set in the category of message types.

3.4.1 Connecting Categorical and Operational Semantics

The categorical semantics developed above relate to the π_{JAPL} operational semantics (Theorem 3.4) through a denotational–operational bridge. Every well-typed π_{JAPL} process P with mailbox type A induces a process presheaf $[\![P]\!] \in \mathbf{TProc}$, where $[\![P]\!](t)$ is the set of reachable states of P after exactly t reduction steps. The restriction maps $[\![P]\!](m \le n) : [\![P]\!](n) \to [\![P]\!](m)$ are well-defined because every state reachable in n steps determines (by truncating the reduction trace) the state at step m. *Soundness* asserts that if two processes are identified in the categorical model—i.e., their presheaves are naturally isomorphic in **TProc**—then they are typed-bisimilar in the operational semantics: $[\![P]\!] \cong [\![Q]\!]$ implies $(P, A) \sim (Q, A)$. This follows because a natural isomorphism between presheaves preserves the recv transition structure at every time step, which is exactly the transfer property required by typed bisimulation (Theorem 3.11). *Completeness* (the converse) holds for finite-state processes: if $(P, A) \sim (Q, A)$, the bisimulation relation itself provides the components of a natural isomorphism $[\![P]\!] \cong [\![Q]\!]$. For infinite-state processes, completeness requires passing to the coalgebraic completion of **TProc**, which we leave to future work. Together, these results ensure that reasoning in the categorical model faithfully reflects the operational behavior of π_{JAPL} programs.

3.5 Bisimulation

Definition 3.11 (Typed bisimulation)**.** *A* typed bisimulation *is a relation $\mathcal{R}$ on typed processes such that if $(P, A)\mathcal{R}(Q, A)$ (note: same message type), then:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $(P', A)\mathcal{R}(Q', A)$.*
2. *If $P \xrightarrow{\mathsf{receive}\, v} P'$ where $v : A$, then there exists Q' such that $Q \xrightarrow{\mathsf{receive}\, v} Q'$ and $(P', A)\mathcal{R}(Q', A)$.*
3. *Symmetrically for Q.*

Two typed processes are bisimilar*, written $(P, A) \sim (Q, A)$, if there exists a typed bisimulation relating them.*

Theorem 3.12 (Congruence)**.** *Typed bisimilarity is a congruence with respect to all π_{JAPL} constructors: parallel composition, restriction, supervision, and spawning.*

Proof sketch. The proof proceeds by showing that for each constructor $\mathcal{C}$, if $(P, A) \sim (Q, A)$ then $(\mathcal{C}[P], A') \sim (\mathcal{C}[Q], A')$ for the appropriate type A'. The key insight is that typed bisimulation respects message types: substituting bisimilar processes in any context preserves the ability to send and receive the same typed messages. The supervision case requires showing that if $P \sim Q$ and both crash with the same typed reason, then their restarts (under any strategy) are also bisimilar. □

4 JAPL's Process Model

JAPL's concurrency model is built on a single primitive: the *process*. A process is a lightweight, isolated unit of concurrent execution with the following properties:

1. **Isolated state:** Each process has its own heap. There is no shared mutable memory between processes. Pure immutable values may be shared (they are safe to share because they cannot change), but mutable resources are owned by exactly one process at a time.

2. **Typed mailbox:** Each process has a single mailbox that accepts messages of a statically known type. The type system prevents sending a message of the wrong type.

3. **Supervision relationship:** Every process (except the root) has a supervisor. If a process crashes, its supervisor is notified and takes corrective action according to a declared strategy.

4. **Location transparency:** Process identifiers ($\mathsf{Pid}[A]$) are location-transparent. A PID may refer to a process on the local node or a remote node; the runtime handles serialization, routing, and network transport.

5. **Effect tracking:** Process operations (spawn, send, receive) are tracked by the effect system. A function that performs process operations must declare the $\mathsf{Process}[A]$ effect in its signature.

4.1 Process Isolation

The cornerstone of JAPL's concurrency safety is process isolation. Unlike Go (where goroutines share memory), Java (where threads share the heap), or even Akka (where JVM actors can access shared objects), JAPL enforces isolation through a combination of language semantics and runtime architecture:

Listing 1: Process isolation: no shared mutable state

```
-- Each process has its own state, managed via recursion
fn worker(state: WorkerState) -> Never with Process[WorkerMsg] =
  match Process.receive() with
  | DoWork(task, reply) ->
      -- state is local to this process; no other process can access
          it
      let result = execute_task(state, task)
      let new_state = update_state(state, task, result)
      Reply.send(reply, result)
      worker(new_state)  -- tail-recursive loop with new state
  | Shutdown ->
      cleanup(state)
      Process.exit(Normal)
```

The key mechanisms ensuring isolation are:

Immutable sharing. Pure values (algebraic data types, records, strings, lists) are immutable and can be shared across process boundaries without copying or synchronization. Since they cannot change, concurrent access is inherently safe.

Ownership transfer. Mutable resources (file handles, buffers, network sockets) are governed by JAPL's ownership type system. A resource has exactly one owner. To pass a mutable resource to another process, ownership must be explicitly transferred, at which point the sending process can no longer access it:

Listing 2: Ownership transfer between processes

```
fn send_to_worker(buf: own Buffer, pid: Pid[WorkerMsg]) -> Unit
    =
  Process.send(pid, ProcessBuffer(buf))
  -- buf is moved; using it here is a compile error
```

Per-process heaps. The runtime allocates a separate heap for each process. Garbage collection of one process's heap does not pause other processes. When a process terminates, its entire heap is reclaimed instantly.

4.2 The Process Effect

Concurrency in JAPL is not ambient: it is tracked by the effect system. Any function that spawns processes, sends messages, or receives messages must declare the Process$[A]$ effect:

Listing 3: The Process effect in function signatures

```
-- Pure function: no concurrency
fn calculate_price(item: Item, qty: Int) -> Money =
  Money.multiply(item.price, qty)

-- Process function: concurrency is declared
fn order_processor(state: State) -> Never with Process[OrderMsg] =
  match Process.receive() with
  | NewOrder(order) ->
      let priced = List.map(order.items, fn item ->
        { item | total = calculate_price(item, item.qty) }
      )
      order_processor({ state | pending = priced })
```

This has several benefits:

- Pure functions are guaranteed to be free of concurrency effects. They can be tested, reasoned about, and composed without concern for race conditions.
- The effect system ensures that process operations cannot "leak" into pure code.
- Effect polymorphism allows writing generic code that works in both pure and concurrent contexts.

4.3 Message Passing Semantics

JAPL's message passing is *asynchronous* and *reliable* for local processes:

Asynchronous send. Process.send(pid, msg) enqueues msg in the target process's mailbox and returns immediately. It never blocks the sender.

Blocking receive. Process.receive() blocks the calling process until a message is available in its mailbox. A timeout variant, Process.receive_with_timeout(ms), returns Err(Timeout) if no message arrives within the specified duration.

Selective receive. Process.receive_matching($predicate$) receives the first message in the mailbox that satisfies the predicate, leaving non-matching messages in the queue. This is essential for implementing protocols where messages may arrive out of order.

Ordering guarantee. Messages from a single sender to a single receiver are delivered in the order they were sent. No ordering is guaranteed between messages from different senders.

Listing 4: Message passing primitives

```
type Msg =
  | StartJob(JobId)
  | CancelJob(JobId)
  | JobFinished(JobId, Output)
  | JobFailed(JobId, Reason)

fn worker_loop(state: WorkerState) -> Process<Unit> {
  receive {
    StartJob(id) =>
      worker_loop(begin_job(state, id))
    CancelJob(id) =>
      worker_loop(cancel_job(state, id))
    Shutdown =>
      done()
  }
}
```

5 Typed Mailboxes

5.1 Motivation

In Erlang, any term can be sent to any process. The burden of message discrimination falls entirely on the receiving process, which must pattern-match on incoming messages and (hopefully) handle all possible cases. This design has served Erlang well for dynamically-typed, hot-code-reloading scenarios, but it has significant drawbacks:

1. **Protocol violations are silent.** Sending a message of the wrong type to a process does not cause an immediate error. The message sits in the mailbox, potentially forever (if no clause matches it), causing a slow memory leak and eventual process crash.

2. **Refactoring is dangerous.** Adding or removing a variant from a message type requires updating all senders—but the compiler provides no guidance on which senders to update.

3. **Documentation is informal.** The expected message protocol of a process is documented in comments or convention, not in the type system.

5.2 JAPL's Approach

In JAPL, every process has a mailbox typed by an algebraic data type. The type of a process identifier, Pid[A], encodes the type of messages the process accepts:

Listing 5: Typed mailbox definition and use

```
type LoggerMsg =
  | Log(Level, String)
  | Flush(Reply[Unit])
  | SetLevel(Level)

-- This process can ONLY receive LoggerMsg values
fn logger(config: LogConfig) -> Never with Process[LoggerMsg] =
  match Process.receive() with
  | Log(level, msg) ->
      if level >= config.min_level then
        write_log(config.output, level, msg)
      logger(config)
  | Flush(reply) ->
      flush_output(config.output)
      Reply.send(reply, ())
      logger(config)
  | SetLevel(level) ->
      logger({ config | min_level = level })

-- Spawning returns a typed PID
let pid: Pid[LoggerMsg] = Process.spawn(fn -> logger(default_config))

-- Type-safe sending: only LoggerMsg values are accepted
Process.send(pid, Log(Info, "server started"))
Process.send(pid, SetLevel(Debug))

-- Type error: wrong message type
-- Process.send(pid, "hello")      -- COMPILE ERROR
-- Process.send(pid, Increment)    -- COMPILE ERROR
```

5.3 Contrast with Erlang

Consider the equivalent Erlang code:

Listing 6: Erlang: untyped messages

```
logger(Config) ->
  receive
    {log, Level, Msg} ->
      case Level >= maps:get(min_level, Config) of
        true -> write_log(maps:get(output, Config), Level, Msg);
        false -> ok
      end,
      logger(Config);
    {flush, From} ->
      flush_output(maps:get(output, Config)),
      From ! ok,
      logger(Config);
    {set_level, Level} ->
      logger(Config#{min_level => Level})
    %% What happens if someone sends an unexpected message?
    %% It sits in the mailbox forever, leaking memory.
  end.
```

In Erlang, there is no compile-time check that senders use the correct tuple structure. A misspelling (`{logmsg, info, "text"}` instead of `{log, info, "text"}`) creates a silent

failure.

5.4 Reply Channels and Request-Response

A common pattern in concurrent systems is request-response: a client sends a request and expects a reply. JAPL supports this through the Reply[T] type, which is a one-shot, typed reply channel:

Listing 7: Request-response with typed reply channels

```
type CacheMsg =
  | Get(Key, Reply[Option[Value]])
  | Put(Key, Value)
  | Invalidate(Key)
  | Stats(Reply[CacheStats])

fn cache_server(store: Map[Key, Value]) -> Never with Process[
    CacheMsg] =
  match Process.receive() with
  | Get(key, reply) ->
      Reply.send(reply, Map.lookup(store, key))
      cache_server(store)
  | Put(key, value) ->
      cache_server(Map.insert(store, key, value))
  | Invalidate(key) ->
      cache_server(Map.delete(store, key))
  | Stats(reply) ->
      Reply.send(reply, compute_stats(store))
      cache_server(store)

-- Client-side: type-safe request-response
fn lookup_cached(pid: Pid[CacheMsg], key: Key) -> Option[Value] with
     Process =
  let reply = Reply.new()
  Process.send(pid, Get(key, reply))
  Reply.await(reply)  -- blocks until reply arrives; typed as Option
      [Value]
```

The Reply[T] type is *affine* in implementation: it may be used *at most once*, with a complementary liveness check that ensures it is used *at least once* on every reachable code path. Together, these two checks enforce exactly-once semantics. JAPL does not require a full substructural type system; instead, Reply[T] values are tracked by a lightweight affine usage analysis integrated into the type checker. Each Reply[T] binding is marked as "unconsumed" at creation and "consumed" after a call to Reply.send. A second use after consumption, or a path where the binding is never consumed, triggers a compile-time error. The following example illustrates the compile-time rejection of double use:

Listing 8: Compile error: Reply used twice

```
fn bad_handler(state: State) -> Never with Process[CacheMsg] =
  match Process.receive() with
  | Get(key, reply) ->
      Reply.send(reply, Map.lookup(state.store, key))
      -- COMPILE ERROR: linear value 'reply' used after consumption
      Reply.send(reply, None)
      bad_handler(state)
```

Similarly, forgetting to reply in a branch is rejected:

Listing 9: Compile error: Reply not consumed on all paths

```
fn forgetful_handler(state: State) -> Never with Process[CacheMsg] =
  match Process.receive() with
  | Get(key, reply) ->
      if Map.contains(state.store, key) then
        Reply.send(reply, Map.lookup(state.store, key))
        -- 'reply' consumed here
      else
        ()  -- COMPILE ERROR: linear value 'reply' not consumed on
            this path
      forgetful_handler(state)
```

This approach prevents a common class of bugs in actor systems—unanswered requests and duplicate replies—without imposing the full cognitive overhead of a substructural type system on all values.

5.5 Session Types for Complex Protocols

For protocols more complex than single request-response, JAPL supports session types [Honda, 1993] that describe multi-step communication sequences:

Listing 10: Session-typed protocol

```
type AuthProtocol =
  session {
    client -> server: Credentials
    server -> client: AuthResult
    if authenticated:
      client -> server: Request
      server -> client: Response
    else:
      end
  }

-- The compiler verifies that both sides follow the protocol
fn auth_client(server: Pid[AuthProtocol.Server]) -> Result[Response,
    AuthError]
    with Process =
  let session = Session.connect(server)
  Session.send(session, my_credentials)
  match Session.receive(session) with
  | Authenticated(token) ->
      Session.send(session, my_request)
      let response = Session.receive(session)
      Ok(response)
  | Rejected(reason) ->
      Err(AuthFailed(reason))
```

Session types provide compile-time guarantees that:

- Messages are sent and received in the correct order.
- Both parties agree on the types of messages at each step.
- All protocol branches are handled.
- Sessions are completed (no dangling sessions).

5.6 Formal Typing of Mailboxes

Definition 5.1 (Mailbox typing rule)**.** *A process P with mailbox type A is well-typed, written $\Gamma \vdash_A P$, if:*

1. *Every* $\mathsf{receiveself}y.Q$ *in P binds y at type A.*
2. *Every pattern match on a received message y is exhaustive over the constructors of A.*
3. *Every* $\mathsf{send}xv.Q$ *where* $\Gamma(x) = \mathsf{Pid}[B]$ *has* $v : B$*.*

Theorem 5.2 (Typed message safety)**.** *If $\Gamma \vdash P$ in π_{JAPL}, then during any execution of P, every message received by a process with mailbox type A has type A.*

Proof. By induction on the typing derivation and case analysis on the reduction rules. The key cases are:

Case R-Comm: The communication rule $\mathsf{send}xv.P \parallel \mathsf{receive}xy.Q \to P \parallel Q[v/y]$ requires, by T-Send, that $v : A$ where $\Gamma(x) = \mathsf{Pid}[A]$, and by T-Recv, that y is bound at type A. Therefore $Q[v/y]$ is well-typed.

Case R-Spawn: The spawn rule creates a new channel $x : \mathsf{Pid}[A]$. By T-Spawn, the spawned process has effect $\mathsf{Process}[A]$, so it receives messages of type A. The continuation can only send type-A messages to x.

Case R-Par: By induction hypothesis on each component.

Case R-New: By induction hypothesis under the extended context.

The supervision case (R-Fail) preserves typing because the restart creates a fresh instance of the same type. □

6 Supervision Trees

6.1 Philosophy: Let It Crash

Erlang's "let it crash" philosophy [Armstrong, 2003] is one of the most important insights in the history of concurrent programming. The idea is simple but profound: instead of trying to anticipate and handle every possible failure within a process, allow the process to crash and delegate recovery to a supervisor. This approach has several advantages:

1. **Simplicity.** Process code handles the "happy path." It does not need defensive error handling for every possible corruption of state, hardware failure, or unexpected condition.
2. **Clean state.** Restarting a process gives it fresh, known-good state. This eliminates the problem of processes continuing with corrupted state after a partial failure.
3. **Isolation of failure.** A crash in one process does not corrupt the state of other processes. Supervision boundaries are fault boundaries.
4. **Compositionality.** Supervision trees compose: a supervisor can itself be supervised, creating hierarchical fault domains.

JAPL adopts this philosophy and extends it with static typing.

6.2 Supervision Strategies

JAPL provides three supervision strategies, matching Erlang/OTP:

`OneForOne:` When a child process crashes, only that child is restarted. Other children are unaffected. This is appropriate when children are independent.

`AllForOne`: When any child crashes, all children are terminated and restarted. This is appropriate when children have complex interdependencies and their collective state must be consistent.

`RestForOne`: When a child crashes, that child and all children started *after* it are terminated and restarted. This is appropriate when children have a sequential dependency: later children depend on earlier ones.

Listing 11: Supervisor declaration with typed crash reasons

```
supervisor ApiSystem {
  strategy: OneForOne
  child http_server()
  child metrics_reporter()
  child cache_manager()
}

-- Expanded form with full child specifications
fn start_app() -> Pid[SupervisorMsg] with Process =
  Supervisor.start(
    strategy = OneForOne,
    max_restarts = 5,
    max_seconds = 60,
    children = [
      { id = "db_pool"
      , start = fn -> DbPool.start(config.database)
      , restart = Permanent
      , shutdown = Timeout(5000)
      },
      { id = "http_server"
      , start = fn -> HttpServer.start(config.http)
      , restart = Permanent
      , shutdown = Timeout(10000)
      },
      { id = "background_jobs"
      , start = fn -> JobRunner.start(config.jobs)
      , restart = Transient
      , shutdown = Timeout(30000)
      },
    ]
  )
```

6.3 Restart Policies

Each child process has a restart policy:

`Permanent`: Always restarted when it terminates, regardless of reason. Used for long-running services.

`Transient`: Restarted only if it terminates abnormally (crashes). Normal termination is not restarted. Used for tasks that may complete.

`Temporary`: Never restarted. Used for one-shot tasks.

6.4 Restart Intensity and Backoff

To prevent restart storms (where a persistently failing process is restarted in a tight loop, consuming resources), supervisors enforce a *restart intensity*: a maximum number of restarts within a time window. If the intensity is exceeded, the supervisor itself crashes, propagating the failure up the supervision tree.

Listing 12: Declarative backoff for restart intensity

```
supervisor DatabaseCluster {
  strategy: OneForOne
  max_restarts: 5
  max_seconds: 60
  backoff: Exponential { base_ms = 100, max_ms = 30000, jitter =
      True }

  child db_connection("primary", config.primary_url)
  child db_connection("replica-1", config.replica1_url)
  child db_connection("replica-2", config.replica2_url)
}
```

The `backoff` specification is a JAPL extension to the OTP model. Rather than immediately restarting a failed child, the supervisor waits for an exponentially increasing duration, with optional jitter to prevent thundering-herd effects. This is particularly valuable for processes that connect to external resources (databases, APIs), where immediate restart after a connection failure is unlikely to succeed.

6.5 Typed Crash Reasons

In Erlang, crash reasons are arbitrary terms. In JAPL, crash reasons are typed, enabling compile-time reasoning about failure modes:

Listing 13: Typed crash reasons

```
type DbCrashReason =
  | ConnectionLost(String)
  | QueryTimeout(Duration)
  | ProtocolError(Bytes)
  | AuthenticationFailed

type HttpCrashReason =
  | BindFailed(Port)
  | TlsError(TlsErrorKind)
  | AcceptError(IoError)

-- Supervisor can pattern-match on typed crash reasons
fn on_child_crash(child_id: String, reason: CrashReason) ->
    SupervisorAction =
  match reason with
  | DbCrashReason(ConnectionLost(_)) ->
      Restart(Backoff(Exponential { base_ms = 500 }))
  | DbCrashReason(AuthenticationFailed) ->
      -- Don't restart; auth won't fix itself
      Escalate
  | HttpCrashReason(BindFailed(port)) ->
      Log.error("Port " ++ show(port) ++ " unavailable")
      Escalate
  | _ ->
      RestartImmediately
```

6.6 Supervision Tree Visualization

The supervision tree structure for a typical JAPL application is illustrated in Figure 2.

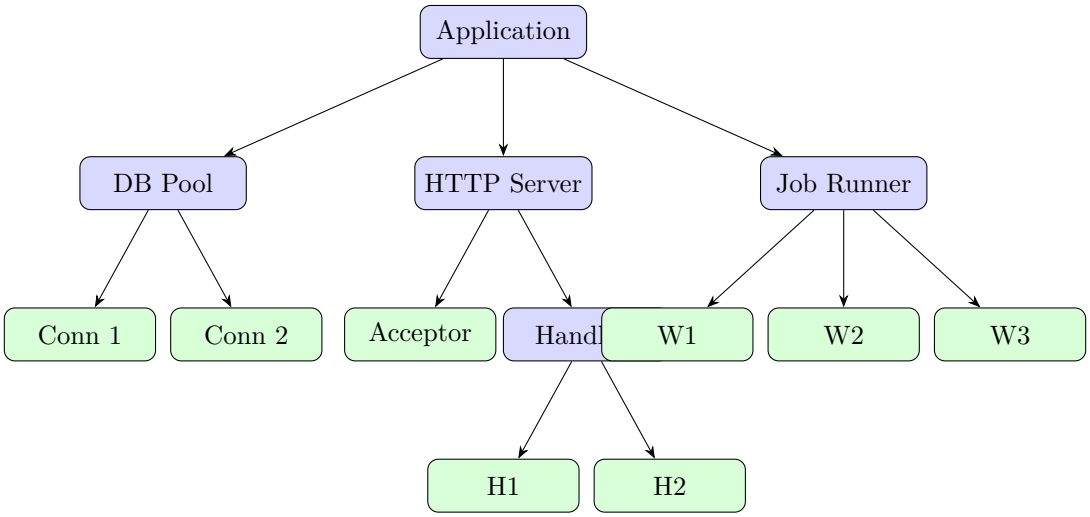

Figure 2: A supervision tree for a typical JAPL web application. Blue nodes are supervisors; green nodes are worker processes. Arrows indicate supervision relationships.

6.7 Process Groups and Registries

Beyond tree-structured supervision, JAPL provides process groups and registries for organizing processes:

Listing 14: Process groups and registries

```
-- Register a process under a name
Registry.register("cache_server", pid)

-- Look up a process by name
let cache = Registry.lookup("cache_server")

-- Process groups: broadcast to all members
let group = ProcessGroup.new("event_listeners")
ProcessGroup.join(group, listener1)
ProcessGroup.join(group, listener2)
ProcessGroup.broadcast(group, EventOccurred(event_data))
```

6.8 Formal Model of Supervision

Definition 6.1 (Supervision tree)**.** *A supervision tree $\mathcal{T}$ is a rooted tree where:*

- *Each internal node is a supervisor* $\mathsf{Sups}\overline{P_i}$ *with strategy* $s \in \{\mathit{OneForOne}, \mathit{AllForOne}, \mathit{RestForOne}\}$.
- *Each leaf is a worker process* P.
- *Each node has restart parameters* (k, t) *specifying at most* k *restarts in* t *seconds.*

Definition 6.2 (Restart function)**.** *The restart function* $\mathsf{restart} : \mathsf{Strategy} \times \mathsf{Reason} \times \overline{\mathsf{Proc}} \times \mathbb{N} \to \overline{\mathsf{Proc}} + \mathsf{Escalate}$ *is defined by:*

$$\mathsf{restart}(\mathit{OneForOne}, r, [P_1, \ldots, P_n], j) = [P_1, \ldots, P_j^0, \ldots, P_n]$$
$$\mathsf{restart}(\mathit{AllForOne}, r, [P_1, \ldots, P_n], j) = [P_1^0, \ldots, P_n^0]$$
$$\mathsf{restart}(\mathit{RestForOne}, r, [P_1, \ldots, P_n], j) = [P_1, \ldots, P_{j-1}, P_j^0, \ldots, P_n^0]$$

where P_i^0 denotes the initial state of process P_i and j is the index of the failed process. If the restart intensity is exceeded, the function returns Escalate*, causing the supervisor itself to fail.*

7 Process Lifecycle

7.1 Lifecycle States

A JAPL process passes through the following states:

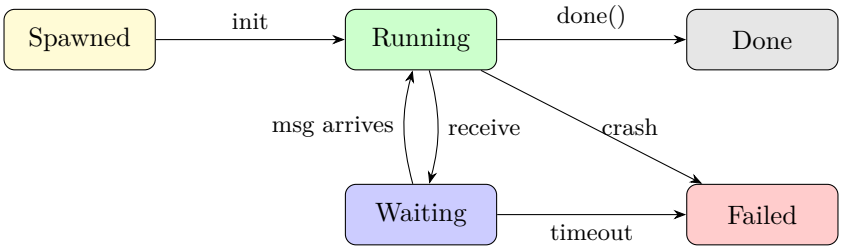

Figure 3: Process lifecycle state machine.

7.2 Lifecycle Operations

The six fundamental process operations are:

spawn: Creates a new process. The parent receives a typed Pid[A] for the new process. The new process begins executing its body function.

Listing 15: Spawning a process

```
let pid: Pid[WorkerMsg] = Process.spawn(fn -> worker(
    initial_state))
```

send: Enqueues a typed message in the target process's mailbox. Non-blocking; returns immediately.

Listing 16: Sending a message

```
Process.send(pid, StartJob(job_id))
```

receive: Blocks the process until a message is available in its mailbox. The received message is pattern-matched.

Listing 17: Receiving with pattern matching

```
match Process.receive() with
| StartJob(id) -> begin_work(state, id)
| CancelJob(id) -> cancel_work(state, id)
```

continue: A tail-recursive call that advances the process to its next iteration with new state. This is the standard mechanism for process-local state management:

Listing 18: State evolution through tail recursion

```
fn counter(n: Int) -> Never with Process[CounterMsg] =
  match Process.receive() with
  | Increment -> counter(n + 1)       -- continue with new state
  | Decrement -> counter(n - 1)       -- continue with new state
  | GetCount(reply) ->
```

```
        Reply.send(reply, n)
        counter(n)                          -- continue with same
            state
```

done: Signals successful process termination. The process exits with reason Normal. Transient and temporary child processes are not restarted after normal termination.

Listing 19: Normal process termination

```
fn one_shot_task(data: TaskData) -> Unit with Process[TaskMsg] =
  let result = compute(data)
  Process.send(data.reply_to, TaskComplete(result))
  done()
```

fail: Signals abnormal process termination with a typed reason. The supervisor is notified and takes action according to its strategy.

Listing 20: Explicit failure with typed reason

```
fn db_worker(conn: DbConn) -> Never with Process[DbMsg] =
  match Process.receive() with
  | Query(sql, reply) ->
      match Db.execute(conn, sql) with
      | Ok(rows) ->
          Reply.send(reply, Ok(rows))
          db_worker(conn)
      | Err(ConnectionLost(detail)) ->
          fail(DbCrashReason(ConnectionLost(detail)))
```

7.3 Process-Local Tail Recursion as State Management

A distinctive feature of JAPL's process model (shared with Erlang) is that process-local state is managed through tail recursion rather than mutable variables. The process function takes the current state as a parameter and calls itself with the new state:

Listing 21: State management via tail recursion

```
type RateLimiterMsg =
  | Request(ClientId, Reply[Bool])
  | ResetBucket(ClientId)
  | GetStats(Reply[RateLimiterStats])

type BucketState = Map[ClientId, { count: Int, reset_at: Timestamp
    }]

fn rate_limiter(state: BucketState, config: RateLimiterConfig)
    -> Never with Process[RateLimiterMsg] =
  match Process.receive() with
  | Request(client, reply) ->
      let bucket = Map.lookup(state, client)
        |> Option.unwrap_or({ count = 0, reset_at = now() + config.
            window })
      if bucket.count < config.max_requests then
        Reply.send(reply, True)
        let new_bucket = { bucket | count = bucket.count + 1 }
        rate_limiter(Map.insert(state, client, new_bucket), config)
      else
        Reply.send(reply, False)
```

```
        rate_limiter(state, config)
  | ResetBucket(client) ->
      rate_limiter(Map.delete(state, client), config)
  | GetStats(reply) ->
      Reply.send(reply, compute_stats(state))
      rate_limiter(state, config)
```

This approach has important advantages over mutable state:

1. **No partial updates.** The state transition is atomic: the old state is the parameter, the new state is the argument to the recursive call. There is no window where the state is partially updated.

2. **State history.** Since state is passed as a value, it is trivial to log, snapshot, or roll back state transitions for debugging or auditing.

3. **Restart safety.** When a process crashes and is restarted, it begins with fresh, known-good state (the initial parameter). There is no corrupted mutable state to worry about.

4. **Guaranteed tail-call optimization.** JAPL guarantees tail-call elimination, so process loops consume constant stack space regardless of how many messages they process.

8 Comparison with Existing Approaches

Table 1 provides a feature-level comparison of JAPL's process model with the concurrency facilities of Erlang, Go, Akka, Pony, and Gleam.

Table 1: Comparison of concurrency models across languages.

Feature	JAPL	Erlang	Go	Akka	Pony	Gleam
Typed messages	✓	×	✓	Partial	✓	✓
Process isolation	✓	✓	×	×	✓	✓
Supervision trees	✓	✓	×	✓	×	✓
Typed supervision	✓	×	×	Partial	×	×
Session types	✓	×	×	×	×	×
Effect tracking	✓	×	×	×	×	×
Location transparency	✓	✓	×	✓	×	✓
Hot code upgrade	Planned	✓	×	×	×	✓
Per-process GC	✓	✓	×	×	✓	✓
Linear resources	✓	×	×	×	✓	×

Note on hot code upgrade. The "Planned" status for hot code upgrade in JAPL reflects a deliberate design tension. Erlang and Gleam support hot code swapping via the BEAM's module-loading infrastructure, which benefits from dynamic typing: a new module version can accept messages of any shape, and the programmer is responsible for handling both old- and new-format messages during the transition. In a statically typed language like JAPL, hot-swapping is *harder* because type versioning must be addressed: if version n of a process expects Msg_n and version $n{+}1$ expects Msg_{n+1}, the runtime must ensure that in-flight messages of type Msg_n are either drained before the upgrade or translated to Msg_{n+1} via a compiler-generated migration function. Conversely, static typing makes hot-swapping *safer*: the compiler can verify that the migration function is total and type-correct, that the new process state type is compatible with the serialized state of the old version, and that all callers holding a $\mathsf{Pid}[\mathsf{Msg}_n]$ are updated or provided with a typed adapter. JAPL's planned approach requires each hot-upgradeable module to declare a $\mathsf{Migration}[V_{\text{old}}, V_{\text{new}}]$ instance, providing compile-time

assurance that upgrades preserve type safety—a guarantee that neither Erlang nor Gleam can offer.

8.1 Erlang/OTP

Erlang is the direct inspiration for JAPL's process model, and JAPL inherits its core insights: lightweight processes, process isolation, asynchronous message passing, supervision trees, and the "let it crash" philosophy. The key differences are:

Static typing. JAPL's typed mailboxes catch message-protocol errors at compile time. Erlang's Dialyzer provides success typing but cannot enforce exhaustive message handling or session-type compliance.

Typed crash reasons. In Erlang, crash reasons are arbitrary terms. In JAPL, they are typed algebraic data types, enabling pattern-matching on failure modes in supervision logic.

Effect tracking. JAPL's effect system makes concurrency operations visible in function signatures. In Erlang, any function can send messages or spawn processes, and this is not reflected in the function's type.

Ownership for resources. Erlang relies on GC for all memory management. JAPL's linear types provide deterministic resource cleanup and safe ownership transfer between processes.

Erlang's advantages over JAPL include its mature ecosystem, battle-tested runtime (the BEAM), hot code reloading, and three decades of production experience. JAPL aims to preserve Erlang's strengths while adding the benefits of static typing.

8.2 Go

Go's goroutines and channels provide syntactically lightweight concurrency, but the model differs from JAPL's in fundamental ways:

Shared memory. Go goroutines share the process memory space. While channels provide a communication mechanism, nothing prevents goroutines from accessing shared variables, and data races are a common source of bugs. Go's race detector is dynamic, not static.

No supervision. Go has no built-in supervision mechanism. A panicking goroutine crashes the program unless explicitly recovered. Building supervision in Go requires manual bookkeeping.

Channel typing. Go channels are typed, which is a strength. However, channels are point-to-point, and there is no equivalent of Erlang/JAPL's selective receive or mailbox semantics. Complex protocols require careful channel management.

Goroutine leaks. Since goroutines are not supervised, a goroutine that blocks on a channel read forever is a resource leak. Detecting and preventing goroutine leaks requires discipline and tooling external to the language.

Go's advantages include its simplicity, fast compilation, excellent tooling, and large ecosystem. JAPL draws inspiration from Go's pragmatic approach to tooling and compilation speed.

8.3 Akka

Akka provides actor-based concurrency on the JVM, with supervision trees and location transparency. The comparison with JAPL:

JVM overhead. Akka actors share the JVM heap and can access each other's state through Java references, bypassing the actor abstraction. JAPL enforces isolation at the language level.

Akka Typed. Akka Typed addresses some message-typing concerns but operates within Java's type system, which lacks algebraic data types and effect tracking. Protocol violations can still occur through type erasure and runtime casts.

Complexity. The Akka API surface is large, with many ways to accomplish the same task. JAPL aims for a smaller, more principled set of primitives.

Library vs. language. Supervision in Akka is a library feature. In JAPL, it is a language feature with syntactic support (the `supervisor` declaration) and type-system integration.

8.4 Pony

Pony's reference capabilities provide a novel approach to actor safety:

Reference capabilities. Pony uses a capability system (iso, val, ref, box, tag, trn) to prevent data races. This is theoretically elegant but imposes significant cognitive burden.

No supervision. Pony does not provide built-in supervision trees. Error handling is left to the programmer.

Actor isolation. Pony achieves actor isolation through its capability system rather than per-actor heaps. This allows zero-copy message passing for data with appropriate capabilities, but requires understanding the capability system.

JAPL takes a different approach: isolation through per-process heaps and ownership types, which is conceptually simpler (immutable data is shared freely; mutable resources have single owners) while achieving similar safety guarantees.

8.5 Gleam

Gleam compiles to BEAM bytecode and provides static typing for the Erlang ecosystem:

Shared runtime. Gleam inherits BEAM's runtime characteristics (per-process heaps, preemptive scheduling, supervision) and can interoperate with Erlang and Elixir libraries.

Type system scope. Gleam's type system is simpler than JAPL's: it lacks effect tracking, session types, linear types, and capability types. This makes Gleam more accessible but provides fewer static guarantees.

Interoperability. Gleam's BEAM compatibility is a significant practical advantage. JAPL targets its own runtime, which provides more control but requires building an ecosystem from scratch.

9 Implementation

This section describes the key implementation techniques for JAPL's process runtime, targeting scalability to millions of concurrent processes on commodity hardware.

9.1 Lightweight Process Representation

Each JAPL process is represented as a lightweight structure containing:

- A **process control block** (PCB): status flags, priority, reduction counter, mailbox pointer, links/monitors list, supervisor reference.
- A **private heap**: a small, separately collected memory region for the process's local data. Initial size is 2 KB, growing as needed.
- A **stack**: for the process's continuation. Since JAPL guarantees tail-call elimination for process loops, the stack remains small.
- A **mailbox**: a queue of incoming messages (described in Section 9.3).

The total overhead per idle process is approximately 4–8 KB, enabling millions of processes per node on machines with tens of gigabytes of RAM.

9.2 Scheduler Design

JAPL's scheduler follows the Erlang BEAM model with enhancements:

Scheduler threads. The runtime creates one scheduler thread per CPU core. Each scheduler thread has a run queue of processes ready to execute.

Preemptive scheduling. Each process is assigned a *reduction budget* (e.g., 4000 reductions). Function calls, message sends, and receives each consume one or more reductions. When a process exhausts its budget, it is preempted and placed at the back of its scheduler's run queue.

Work stealing. When a scheduler's run queue is empty, it attempts to steal processes from other schedulers' queues. This provides automatic load balancing across cores.

Priority queues. Each scheduler maintains multiple priority levels (high, normal, low). High-priority processes (e.g., supervisors, timers) are scheduled preferentially.

Priority inversion avoidance. In shared-memory systems, priority inversion occurs when a low-priority thread holds a lock needed by a high-priority thread. JAPL's process model avoids this pathology structurally: since processes do not share memory or locks, the classical Coffman-style priority inversion cannot arise. However, a *message-level* analog exists: a high-priority process may send a request (via $\mathsf{Reply}[T]$) to a low-priority process and block on the reply. If the low-priority process is starved of scheduler time, the high-priority process is effectively delayed. JAPL addresses this through *priority inheritance on reply channels*: when a high-priority process blocks on a Reply.await, the runtime temporarily elevates the priority of the target process to match the waiter's priority. This elevation persists until the reply is sent, at which point the target process reverts to its original priority. During work stealing, stolen processes retain their (possibly elevated) priority, ensuring that a process promoted due to a pending reply is not demoted simply by being migrated to a different scheduler's queue. This mechanism ensures that priority queues and work stealing interact correctly without reintroducing the priority inversion problem.

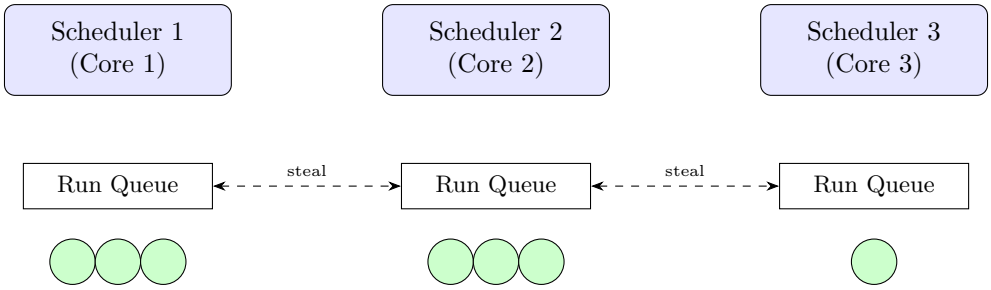

Figure 4: Work-stealing scheduler architecture. Each core has a scheduler with its own run queue. Idle schedulers steal work from busy ones.

9.3 Mailbox Data Structures

The mailbox is a critical data structure, as every inter-process communication passes through it. JAPL's mailbox implementation has two components:

Message queue. A lock-free, multi-producer, single-consumer (MPSC) queue. Multiple processes can send messages concurrently without contention (each sender appends to the queue via a CAS operation on the tail pointer). The owning process is the sole consumer, so receives require no synchronization.

Selective receive index. For processes that use selective receive, the mailbox maintains an auxiliary index that maps message constructors (tags) to positions in the queue. This enables $O(1)$ lookup for selective receive by tag, avoiding the $O(n)$ scan that Erlang's selective receive requires.

Listing 22: Selective receive with efficient tag-based lookup

```
-- Without selective receive: messages processed in order
fn simple_worker(state: State) -> Never with Process[WorkerMsg] =
  match Process.receive() with
  | msg -> handle(state, msg)

-- With selective receive: priority messages handled first
fn priority_worker(state: State) -> Never with Process[WorkerMsg] =
  -- Efficiently finds the first Priority message in the mailbox
  match Process.receive_matching(fn m -> is_priority(m)) with
  | Some(Priority(data)) -> handle_priority(state, data)
  | None ->
      -- No priority messages; handle the next regular message
      match Process.receive() with
      | msg -> handle(state, msg)
```

9.4 Process Isolation Guarantees

JAPL's runtime enforces process isolation through multiple mechanisms:

1. **Per-process heaps.** Each process allocates memory from its own heap region. References between heaps are forbidden for mutable data.

2. **Immutable data sharing.** Immutable values (which constitute the vast majority of data in functional programs) can be shared across process heaps via pointer sharing, since concurrent reads of immutable data are safe. The GC handles reachability tracking.

3. **Message copying for small messages.** Small messages (below a configurable threshold, default 64 bytes) are copied into the receiving process's heap. This avoids cross-heap references for small messages and improves cache locality.

4. **Reference counting for large immutable messages.** Large immutable messages (strings, byte arrays, large records) are allocated in a shared immutable heap and reference-counted. Since they are immutable, concurrent reference counting is safe.

5. **Ownership transfer for mutable resources.** Mutable resources are moved (not copied) between processes. The runtime ensures that after a move, the sending process's reference is invalidated.

9.5 Green Threads vs. OS Threads

JAPL processes are green threads (user-space threads managed by the runtime) rather than OS threads. The comparison:

Table 2: Green threads vs. OS threads.

Property	**Green Threads (JAPL)**	**OS Threads**
Creation cost	∼2–8 KB memory	∼1–8 MB stack
Context switch	∼100 ns (user-space)	∼1–10 μs (kernel)
Max concurrent	Millions per node	Thousands per node
Scheduling	Runtime (cooperative/preemptive)	OS kernel
Isolation	Per-process heap (JAPL runtime)	Shared process memory
Blocking I/O	Handled by I/O scheduler	Blocks OS thread

The JAPL runtime includes a dedicated I/O scheduler that offloads blocking operations (file I/O, DNS resolution, long-running FFI calls) to a pool of OS threads, ensuring that blocking I/O does not stall the green thread scheduler.

9.6 Garbage Collection

JAPL uses a hybrid GC strategy:

Per-process generational GC. Each process's private heap has a two-generation collector. Young generation collection is triggered frequently (every few hundred reductions) and is very fast (typically microseconds) because process heaps are small. Old generation collection is less frequent.

Process death reclamation. When a process terminates, its entire heap is freed in one operation, with no need for GC. In systems with many short-lived processes, this provides excellent throughput.

Shared immutable heap. Large immutable values shared across processes live in a separate heap with concurrent mark-sweep collection. Since this data is immutable, no write barriers are needed, simplifying the collector significantly.

10 Formal Properties

10.1 Typed Message Safety

Theorem 10.1 (Type preservation)**.** *If* $\Gamma \vdash P$ *and* $P \to P'$*, then* $\Gamma' \vdash P'$ *for some* $\Gamma' \supseteq \Gamma$*.*

Proof. By induction on the derivation of $P \to P'$.

Case R-Comm: We have $\mathsf{send}xv.P_1 \parallel \mathsf{receive}xy.P_2 \to P_1 \parallel P_2[v/y]$. By T-Send, $\Gamma \vdash v : A$ where $\Gamma(x) = \mathsf{Pid}[A]$. By T-Recv, $\Gamma, y : A \vdash P_2$. By the substitution lemma, $\Gamma \vdash P_2[v/y]$. Since $\Gamma \vdash P_1$ (from the premise of T-Send), we have $\Gamma \vdash P_1 \parallel P_2[v/y]$.

Case R-Spawn: We have $\mathsf{spawn}xAP_1P_2 \to (\nu\, x : \mathsf{Pid}[A])(P_1 \parallel P_2)$. By T-Spawn, $\Gamma, x : \mathsf{Pid}[A] \vdash P_1$ and $\Gamma, x : \mathsf{Pid}[A] \vdash P_2$. Let $\Gamma' = \Gamma, x : \mathsf{Pid}[A]$. Then $\Gamma' \vdash P_1 \parallel P_2$ and $\Gamma \vdash (\nu\, x : \mathsf{Pid}[A])(P_1 \parallel P_2)$ by T-New (with $\Gamma' \supseteq \Gamma$).

Case R-Fail: The supervisor applies the restart function, which creates fresh instances of the same process type. By T-Sup, each child specification includes a start function of the appropriate type, so the restarted processes are well-typed.

The remaining cases (R-Match, R-Par, R-New) follow by straightforward induction. □

Corollary 10.2 (No message type errors at runtime)**.** *In a well-typed JAPL program, a process with mailbox type A never receives a message that is not of type A.*

10.2 Deadlock Freedom

Deadlock in process-based systems can arise when processes form a circular wait on message reception. JAPL's supervision model provides a practical (though not absolute) guarantee against deadlock:

Definition 10.3 (Supervision-disciplined network)**.** *A process network is* supervision-disciplined *if:*

1. *Every process is part of a supervision tree.*
2. *Communication follows the supervision hierarchy: children send messages to their supervisor (or siblings in the same supervision group), and supervisors send messages to their children.*
3. *All receive operations have timeouts at supervision boundaries.*

Theorem 10.4 (Deadlock freedom under supervision discipline)**.** *A supervision-disciplined process network is deadlock-free.*

Proof. Assume for contradiction that a deadlock exists: a cycle $P_1 \to P_2 \to \cdots \to P_k \to P_1$ where each P_i is waiting for a message from P_{i+1} (indices mod k).

By the supervision discipline, communication follows the supervision hierarchy. The supervision tree is acyclic (it is a tree). Therefore, the communication graph is a subgraph of the tree, augmented with sibling edges. In either case, a cycle would require an edge from a descendant to an ancestor, which the discipline forbids for blocking communication.

If any P_i uses a timed receive at a supervision boundary, the timeout will fire, breaking the potential deadlock by causing P_i to take an alternative action (fail, retry, or escalate to its supervisor). □

Remark 10.5. *The supervision discipline is sufficient but not always necessary. Many practical process networks that violate the discipline are still deadlock-free. However, the discipline provides a compositional, statically checkable criterion.*

10.2.1 Lateral Communication and Deadlock Detection

We acknowledge that the supervision-disciplined network assumption is restrictive. In real-world actor systems, *lateral communication*—messages between sibling processes, unrelated process groups, or processes discovered via a Registry—is common and often necessary. Registry-based communication (e.g., `Registry.lookup("service_name")`) introduces edges in the communication graph that do not follow the supervision hierarchy, potentially creating cycles that violate the tree-structured assumption of Theorem 10.4.

JAPL adopts a two-tier strategy to handle this. For *tree-structured communication* (parent–child and sibling messages within a supervision group), the deadlock prevention guarantee of Theorem 10.4 applies statically. For *lateral communication* that escapes the supervision hierarchy, JAPL provides a *deadlock detection* effect:

Listing 23: Deadlock-aware lateral communication

```
-- Lateral send with deadlock detection enabled
fn request_from_peer(peer: Pid[PeerMsg], data: RequestData)
    -> Result[Response, Timeout] with Process =
  let reply = Reply.new()
  Process.send(peer, PeerRequest(data, reply))
  -- Timed await: breaks potential deadlock cycles
  Reply.await_timeout(reply, 5000)
```

The runtime maintains a lightweight *wait-for graph* over processes that are blocked on Reply.await or receive calls involving lateral channels. When a cycle is detected in this graph, the runtime delivers a DeadlockDetected signal to the youngest process in the cycle (by process creation time), causing it to fail and be restarted by its supervisor. This hybrid approach—static prevention for hierarchical communication, dynamic detection for lateral communication—preserves the strong guarantees of the supervision discipline where they apply while permitting the flexible communication topologies that practical systems require.

10.3 Progress Guarantees

Definition 10.6 (Progress)**.** *A well-typed process P with mailbox type A* makes progress *if it is either:*

1. *Able to perform an internal reduction ($P \to P'$), or*
2. *Waiting to receive a message and able to accept any message of type A, or*
3. *Successfully terminated ($P = \mathsf{done}$), or*
4. *Failed and being handled by its supervisor.*

Theorem 10.7 (Progress for well-typed processes)**.** *If $\Gamma \vdash P$ and $P \neq \mathsf{done}$ and P is not waiting for a message, then there exists P' such that $P \to P'$.*

Proof. By induction on the typing derivation of P.

Case T-Send: $P = \mathsf{send}xv.P'$. If there exists a process $Q = \mathsf{receive}xy.Q'$ in parallel, then R-Comm applies. If not, the send is asynchronous: the message is enqueued and P transitions to P'.

Case T-Match: $P = \mathsf{match}\ v\ \{\overline{p_i \Rightarrow P_i}\}$. Since the typing rule requires exhaustive pattern matching over all constructors of the message type A, there exists some j such that p_j matches v, and R-Match applies.

Case T-Spawn: R-Spawn applies immediately.

Case T-Sup: The supervisor monitors its children. If any child has failed, R-Fail applies. Otherwise, the supervisor itself is waiting for a child event (a form of receive). □

10.4 Typed Reply Safety

Theorem 10.8 (Reply linearity)**.** *If $\Gamma \vdash P$ and P contains a binding $r : \mathsf{Reply}[T]$, then r is used exactly once in P: exactly one $\mathsf{Reply.send}(r, v)$ occurs on every execution path through P.*

Proof. The $\mathsf{Reply}[T]$ type is linear. JAPL's linear type checker ensures that every linear value is consumed exactly once. If r is unused on some path, the checker reports a "linear value not consumed" error. If r is used twice, the checker reports a "linear value used after consumption" error. Since the program is well-typed, neither case applies. □

11 Discussion

11.1 When Processes Are Overkill

Not every concurrent task warrants a process. For fine-grained data parallelism—mapping a function over a large array, parallel sorting, matrix operations—the overhead of process creation and message passing is unnecessary. JAPL addresses this through two mechanisms:

Parallel combinators. Functions like List.par_map and Array.par_fold provide data-parallel operations that use the scheduler's thread pool directly, without creating processes. These are implemented as work-stealing parallel loops at the runtime level.

Listing 24: Data parallelism without processes

```
-- Parallel map: uses scheduler threads, not processes
let results = List.par_map(large_list, fn item ->
  expensive_computation(item)
)

-- Sequential map: single-threaded
let results = List.map(large_list, fn item ->
  expensive_computation(item)
)
```

Structured concurrency scopes. For medium-grained parallelism (e.g., making several API calls concurrently), JAPL provides structured concurrency within a lexical scope:

Listing 25: Structured concurrency scope

```
fn fetch_dashboard_data(user_id: UserId) -> DashboardData
    with Io, Net =
  -- All three fetches run concurrently; scope waits for all
  let (profile, orders, notifications) = concurrent {
    fetch_profile(user_id),
    fetch_orders(user_id),
    fetch_notifications(user_id),
  }
  { profile, orders, notifications }
```

11.2 Integration with the Effect System

The Process[A] effect interacts with other effects in JAPL's effect system:

Effect subsumption. Process[A] subsumes Async: any asynchronous operation can be expressed as a process operation. The reverse is not true: Async does not imply the ability to spawn processes or receive typed messages.

Effect handlers at process boundaries. When spawning a process, the parent provides effect handlers for the child's effects. For example, a child process that performs Io has its I/O routed through the runtime's I/O scheduler. This is transparent to the child.

Pure process logic. The body of a process loop often calls pure functions for business logic, using processes only for state management and message routing. The effect system makes this separation explicit: pure helper functions have no effects in their signatures.

11.3 Distribution Considerations

JAPL's location-transparent PIDs enable distribution, but introduce challenges:

Serialization. Messages sent to remote processes must be serialized. JAPL uses type-derived serialization: any type with a Serialize trait implementation can cross node boundaries. Types that cannot be serialized (e.g., those containing function closures or mutable resources) cause a compile-time error if used in a context that requires distribution.

Partial failure. Network partitions can cause message loss and process unreachability. JAPL's monitoring system detects node disconnections and delivers NodeDown messages to processes that monitor remote nodes.

Consistency. JAPL does not impose a global consistency model. Processes on different nodes communicate via asynchronous messages, and any consistency guarantees must be built at the application level (e.g., using consensus protocols). The supervision tree model naturally supports the "let it crash and retry" approach to network failures.

11.4 Performance Considerations

The process-based model introduces overhead compared to shared-memory concurrency:

Message copying. While immutable data can be shared by reference, crossing process boundaries for mutable data requires copying or ownership transfer. For most applications, the cost is negligible; for high-throughput data pipelines, ownership transfer (zero-copy) is available.

Context switching. Green thread context switches ($\sim$100 ns) are much cheaper than OS thread context switches ($\sim$1–10 μs) but more expensive than atomic operations ($\sim$10–50 ns). For contended data structures accessed billions of times, lock-free shared memory would be faster. JAPL's position is that such scenarios are rare and can be addressed with the parallel combinator escape hatch.

Memory overhead. Per-process heaps provide isolation but increase memory usage compared to a shared heap. The minimum overhead (4–8 KB per process) is acceptable for most applications but may be significant for systems with tens of millions of processes.

11.5 Limitations and Future Work

Several aspects of JAPL's process model are active areas of research and development:

1. **Multiparty session types.** The current session type support handles two-party sessions. Extending to multiparty session types [Honda et al., 2008] would enable verification of complex distributed protocols involving multiple participants.

2. **Hot code upgrade.** Erlang supports hot code swapping—updating running code without stopping the system. JAPL plans to support this through typed module versioning, where the type system ensures that upgraded code is compatible with existing process states.

3. **Process algebra tooling.** We plan to develop tools that extract process algebra models from JAPL code and verify properties (deadlock freedom, protocol compliance) using model checking techniques.

4. **Hardware acceleration.** For some workloads, mapping processes to hardware threads (via OS threads or SIMD lanes) could provide significant speedups. We are investigating hybrid scheduling strategies.

5. **Formal verification.** Integrating with proof assistants (Coq, Lean) to provide machine-checked proofs of process protocol correctness is a long-term goal.

12 Conclusion

We have presented JAPL's process-based concurrency model, which rejects shared-memory concurrency in favor of isolated, lightweight processes communicating through typed mailboxes under hierarchical supervision. This design, rooted in the Erlang/OTP tradition and extended with modern type theory, provides several advantages over both shared-memory and untyped actor systems:

1. **Data race freedom by construction.** Process isolation eliminates data races without requiring locks, atomics, or complex memory models.

2. **Typed message safety.** Typed mailboxes prevent message-protocol errors at compile time, catching a large class of bugs that are silent in dynamically-typed actor systems.

3. **Compositional fault tolerance.** Supervision trees, integrated into the type system with typed crash reasons and declarative backoff, provide hierarchical fault domains that compose naturally.

4. **Effect-tracked concurrency.** The Process effect makes concurrency visible in function signatures, enabling clean separation of pure logic from concurrent state management.

5. **Formal foundations.** The categorical semantics (presheaf categories over time, profunctors for communication) and typed π-calculus provide rigorous foundations for reasoning about process behavior, with proofs of type preservation, progress, and deadlock freedom under supervision discipline.

The process-based model is not without tradeoffs: it introduces overhead for fine-grained parallelism, requires explicit message passing for communication, and adds complexity for distribution. JAPL addresses these through parallel combinators, structured concurrency scopes, and location-transparent PIDs respectively.

Ultimately, JAPL's thesis is that *the shared-memory model is the wrong default for concurrent programming.* Shared memory makes the common case (independent concurrent tasks) unnecessarily dangerous and the hard case (coordinated concurrent tasks) nearly intractable. Process isolation makes the common case safe by default and the hard case manageable through typed protocols and supervision. Three decades of Erlang/OTP's success in telecommunications, along with the growing adoption of actor-based systems in distributed computing, provide strong empirical support for this thesis. JAPL's contribution is to bring the full power of modern type theory to bear on this proven model.

"Pure functions handle logic, supervised processes handle time and failure, and ownership handles reality."

References

Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

Joe Armstrong. Making reliable distributed systems in the presence of software errors. PhD thesis, Royal Institute of Technology, Stockholm, 2003.

Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78. ACM, 2008.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

Gian Luca Cattani and Glynn Winskel. Presheaf models for concurrency. In *CSL*, volume 1414 of *LNCS*, pages 58–75. Springer, 1997.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *AGERE!*, pages 1–12. ACM, 2015.

Edward G. Coffman, Michael J. Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.

Gleam Team. The Gleam programming language. `https://gleam.run`, 2023.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation from open maps. *Information and Computation*, 127(2):164–185, 1996.

Lightbend. Akka: Build concurrent, distributed, and resilient message-driven applications. `https://akka.io`, 2023.

Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178. ACM, 2006.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, pages 378–391. ACM, 2005.

Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, 1992.

Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

Rob Pike. Concurrency is not parallelism. Talk at Heroku's Waza conference, 2012. `https://go.dev/blog/waza-talk`.

Glenn E. Reeves. What really happened on Mars? `https://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html`, 1997.

Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in Action*. Manning Publications, 2016.

A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *WBIA*, pages 62–71. ACM, 2012.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming. PhD thesis, Keio University, 1998.

Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286. ACM, 2012.

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *S&P*, pages 354–368. IEEE, 2014.

Martin Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. *ICFP*, pages 268–279. ACM, 2000.

Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):30:1–30:29, 2019.

Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *LNCS*, pages 350–369. Springer, 2013.

Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

Glynn Winskel. Event structures. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.

A Extended JAPL Process Examples

A.1 A Complete Worker Pool

The following example demonstrates a complete worker pool implementation using supervision, typed mailboxes, and process lifecycle management:

Listing 26: Worker pool with supervision

```
-- Message types for the pool manager
type PoolMsg =
  | SubmitJob(Job, Reply[JobResult])
  | WorkerDone(WorkerId, JobResult)
  | WorkerCrashed(WorkerId, CrashReason)
  | ScaleUp(Int)
  | ScaleDown(Int)
  | GetPoolStats(Reply[PoolStats])

-- Message types for individual workers
type WorkerMsg =
  | RunJob(Job)
  | Shutdown

-- Worker process
fn worker(id: WorkerId, pool: Pid[PoolMsg]) -> Never with Process[
    WorkerMsg] =
  match Process.receive() with
  | RunJob(job) ->
      let result = execute_job(job)
      Process.send(pool, WorkerDone(id, result))
      worker(id, pool)
  | Shutdown ->
      done()

-- Pool state
type PoolState =
  { workers: Map[WorkerId, Pid[WorkerMsg]]
  , available: List[WorkerId]
  , pending: Queue[(Job, Reply[JobResult])]
  , next_id: Int
  }

-- Pool manager process
fn pool_manager(state: PoolState) -> Never with Process[PoolMsg] =
  match Process.receive() with
  | SubmitJob(job, reply) ->
      match state.available with
      | [worker_id, ..rest] ->
          let pid = Map.get(state.workers, worker_id)
          Process.send(pid, RunJob(job))
          let new_state = { state
            | available = rest
            , pending = Queue.push(state.pending, (job, reply))
          }
          pool_manager(new_state)
      | [] ->
          -- No available workers; queue the job
          let new_state = { state
            | pending = Queue.push(state.pending, (job, reply))
```

```
            }
            pool_manager(new_state)

    | WorkerDone(id, result) ->
        -- Return worker to available pool, dispatch next job
        match Queue.pop(state.pending) with
        | Some((next_job, reply), remaining) ->
            let pid = Map.get(state.workers, id)
            Reply.send(reply, result)
            Process.send(pid, RunJob(next_job))
            pool_manager({ state | pending = remaining })
        | None ->
            pool_manager({ state | available = [id, ..state.available]
                })

    | GetPoolStats(reply) ->
        Reply.send(reply, {
          total_workers = Map.size(state.workers),
          available_workers = List.length(state.available),
          pending_jobs = Queue.length(state.pending),
        })
        pool_manager(state)

-- Supervisor for the pool
supervisor WorkerPool {
  strategy: OneForOne
  max_restarts: 10
  max_seconds: 60

  child pool_manager(initial_pool_state())
}
```

A.2 Distributed Key-Value Store

This example shows a distributed key-value store using JAPL's location-transparent PIDs and process groups:

Listing 27: Distributed key-value store

```
type KvMsg =
  | Put(Key, Value, Reply[Ok])
  | Get(Key, Reply[Option[Value]])
  | Delete(Key, Reply[Ok])
  | Replicate(Key, Value)

fn kv_node(store: Map[Key, Value], peers: List[Pid[KvMsg]])
    -> Never with Process[KvMsg] =
  match Process.receive() with
  | Put(key, value, reply) ->
      let new_store = Map.insert(store, key, value)
      -- Replicate to peers (fire-and-forget)
      List.each(peers, fn peer ->
        Process.send(peer, Replicate(key, value))
      )
      Reply.send(reply, Ok)
      kv_node(new_store, peers)
  | Get(key, reply) ->
      Reply.send(reply, Map.lookup(store, key))
```

```
        kv_node(store, peers)
    | Delete(key, reply) ->
        Reply.send(reply, Ok)
        kv_node(Map.delete(store, key), peers)
    | Replicate(key, value) ->
        kv_node(Map.insert(store, key, value), peers)

-- Start nodes on multiple machines
fn start_cluster(nodes: List[NodeAddr]) -> List[Pid[KvMsg]] with
    Process, Net =
  let pids = List.map(nodes, fn addr ->
    let node = Node.connect(addr)
    Process.spawn_on(node, fn -> kv_node(Map.empty(), []))
  )
  -- Inform each node about its peers
  List.each(pids, fn pid ->
    let peers = List.filter(pids, fn p -> p != pid)
    -- In practice, use a separate control message type
    -- This is simplified for illustration
  )
  pids
```

B Process Algebra Derivations

B.1 Encoding Supervision in the Pi-Calculus

A supervisor with strategy $\mathsf{OneForOne}$ and children $P_1, \ldots, P_n$ is encoded as:

$$\begin{aligned}[\![\mathsf{SupO4O}P_1, \ldots, P_n]\!] = (\nu\, m_1, \ldots, m_n)\Big(& \\ & !m_i(\text{down}).(\text{match down with} \\ & \quad \mid \text{Crash}(r) \Rightarrow [\![P_i]\!] \\ & \quad \mid \text{Normal} \Rightarrow \mathbf{0}) \\ & \parallel [\![P_1]\!] \parallel \cdots \parallel [\![P_n]\!]\Big)\end{aligned}$$

where m_i is a monitor channel for child P_i, and the replicated input on each m_i implements the restart logic: upon receiving a crash notification, the supervisor re-instantiates the child process.

B.2 Bisimulation of Supervised Processes

Lemma B.1 (Restart bisimulation)**.** *Let P be a process and P^0 its initial state. Under OneForOne supervision, a process that crashes and is restarted is bisimilar (from the perspective of external observers) to a process that was never started with the corrupted state:*

$$\mathsf{Sup}\mathit{O4OP}\ \mathit{crashes} \sim \mathsf{Sup}\mathit{O4OP}^0$$

Proof sketch. After the crash and restart, the supervised process is in state P^0, which is exactly the initial state. The supervision wrapper is in its monitoring state, ready to handle the next crash. Since P^0 is the same state in both cases, and the supervisor's state depends only on the restart counter (which is the same up to the current restart count), the two configurations are bisimilar. □

C Session Type Encoding

JAPL's session types can be encoded in the linear π-calculus following the Caires-Pfenning correspondence [Caires and Pfenning, 2010]:

Definition C.1 (Session type encoding)**.** *The session type S is encoded as a linear π-calculus type $[\![S]\!]$:*

$$
\begin{aligned}
[\![\mathsf{Send}[T].S]\!] &= T \otimes [\![S]\!] \\
[\![\mathsf{Recv}[T].S]\!] &= T \multimap [\![S]\!] \\
[\![\mathsf{Choice}\{l_i : S_i\}]\!] &= \bigoplus_i [\![S_i]\!] \\
[\![\mathsf{Branch}\{l_i : S_i\}]\!] &= \&_i [\![S_i]\!] \\
[\![\mathsf{End}]\!] &= \mathbf{1}
\end{aligned}
$$

where $\otimes$ is the tensor (send), $\multimap$ is the lollipop (receive), $\oplus$ is the sum (internal choice), $\&$ is the with (external choice), and $\mathbf{1}$ is the unit (session end).

This encoding ensures that the linearity discipline of the π-calculus enforces the session type protocol: each message is sent/received exactly once, branches are handled exhaustively, and sessions are properly terminated.