Orchestrating Autonomous Supply Chains: A Systems Engineering Report on the Global Logistics Broker in Screeps

The transition from localized, script-based automation to industrial-grade autonomous systems within the Screeps execution environment represents one of the most significant engineering challenges in contemporary distributed simulation. At its core, the Screeps environment is not merely a strategy game but a real-time, resource-constrained distributed operating system where the primary bottleneck is not the in-game currency (energy), but the computational throughput of the Central Processing Unit (CPU) and the overhead of memory serialization. This report provides an exhaustive architectural analysis of the design patterns required to implement a top-tier Global Logistics Broker, focusing on the shift from legacy "Search and Rescue" behaviors—typical of the grgisme/screeps era—to a modern "Command and Control" supply chain managed by stable matching algorithms and energy reservation ledgers.

The Physical Architecture of the Screeps Execution Environment

To engineer a high-performance logistics broker, one must first account for the physical constraints of the server-side infrastructure. User code in Screeps runs within a Node.js virtual machine isolate, often managed by isolated-vm to ensure security and resource accounting. This environment is strictly deterministic and operates on a discrete tick-based lifecycle.
The standard execution model involves two distinct stages: the player script calculation and the command processing phase. During the calculation stage, the server loads the user's script, parses the persistent Memory object, and executes the loop function. All commands issued during this time (e.g., creep.move(), creep.transfer()) are collected into a Redis-based queue for batch processing in the subsequent stage. This separation of intent and execution necessitates an architecture that can accurately predict the state of the world multiple ticks into the future.

The Heap Persistence and Isolate Lifecycle

A critical misunderstanding in early-stage development is the belief that the execution environment is completely stateless between ticks. In reality, the global scope—the V8 Heap—persists across ticks unless the server explicitly tears down the isolate to reclaim resources or the user uploads new code. Modern architectures exploit this persistence by caching massive amounts of data in the global object, bypassing the expensive JSON.parse() costs associated with the Memory object.

Feature	Legacy Approach (grgisme)	Modern Industrial Approach	Impact on Performance
State Storage	Primary reliance on	Primary reliance on	Reduces CPU spent on

Feature	Legacy Approach (grgisme)	Modern Industrial Approach	Impact on Performance
	Memory object.	Global Heap Cache.	JSON serialization.
Data Lifecycle	Data is re-parsed and re-processed every tick.	Persistent class instances updated via refresh().	Minimizes garbage collection and object instantiation.
Memory Limit	2MB hard cap on Memory.	2MB Memory + ~256MB+ Heap + 100MB Segments.	Enables massive historical data and complex cost matrices.
Sensing	Direct Room.find() calls per creep.	Centralized Registry with O(1) lookups.	Eliminates redundant spatial search overhead.

The Serialization Wall and the JSON Tax

The Memory object is not a live database connection but a JSON string stored in MongoDB and indexed via Redis. Each tick, the server charges the user the CPU cost of parsing this string. For an empire with multiple rooms and hundreds of creeps, a 2MB memory file can consume upwards of 15ms of CPU just to enter the loop function. This physical constraint dictates that a top-tier logistics broker must minimize the volume of data stored in Memory, favoring ephemeral heap-based structures for task management and only persisting critical "intent" state.

Forensic Analysis of Legacy Logistics Failures

In the grgisme era of Screeps development, logistics were largely emergent properties of independent agent scripts. A "harvester" role would independently seek a source, and a "carrier" role would independently scan for dropped energy or full containers. This "Search and Rescue" model fails at scale for three primary reasons: redundant sensing, collision-heavy movement, and the "Energy Racing" phenomenon.

The Complexity of Redundant Sensing

In a legacy bot, every hauler performs its own environmental assessment. If a room contains 20 haulers, and each hauler executes room.find(FIND_DROPPED_RESOURCES) to locate energy, the server must perform 20 identical spatial searches. This leads to an O(N \cdot M) complexity where N is the number of haulers and M is the number of objects in the room. In contrast, a modern broker performs a single scan, registering all providers and requesters in a centralized registry, reducing the cost to a manageable O(M) scan followed by O(N) matching.

The Energy Racing Phenomenon

Without a centralized broker, multiple haulers often target the same pile of energy or the same empty structure. This occurs because each hauler makes its decision based on the static state of the world at the start of the tick. If three haulers see a pile of 500 energy, all three may move toward it. Only the first to arrive will successfully pick up the resource; the other two will have wasted dozens of ticks in transit. This "race condition" is the primary cause of logistics inefficiency and can lead to catastrophic failure during combat when Towers run dry despite

ample energy being present in the room.

Monolithic Loop Fragility

Legacy bots typically utilize a monolithic loop that iterates through all creeps sequentially. This structure lacks preemption. If the logistics logic for the first ten haulers consumes the entire CPU budget—perhaps due to a complex pathfinding calculation—the remaining haulers and critical defense structures never execute their logic. The bot effectively "freezes," leading to empire collapse.

The Command-and-Control Paradigm: The Global Logistics Broker

A top-tier logistics system replaces decentralized "Roles" with a centralized Broker. This broker functions as an Inversion of Control (IoC) layer, where structures and resources do not "wait" to be found, but "register" their status as either Providers or Requesters.

Defining the Provider/Requester Pattern

In this architecture, every resource-holding or resource-consuming entity is categorized based on its potential to contribute to or drain from the supply chain.

1. **Providers:** These are entities with energy or resources that are "outputting" into the network. This includes sources (via miners), containers, links, dropped resources, and tombstones.
2. **Requesters:** These are entities that need resources to function. This includes Spawns and Extensions (for creep production), Towers (for defense), and the Controller (for upgrading).
3. **Buffers:** These are high-capacity structures like Storage or Terminals that can act as either providers or requesters depending on the current colony state.

The Transport Request Interface

To standardize communication between the broker and the game objects, a TransportRequest interface is established. This interface ensures that the broker has all necessary data to make an optimal matching decision without needing to query the game objects directly during the matching loop.

Property	Type	Purpose
Target	Object	The ID or reference of the structure or resource.
Amount	Number	The total volume of resources requested or provided.
ResourceType	Constant	The type of resource (e.g., RESOURCE_ENERGY).
Priority	Number	A weighted value indicating the urgency of the request.
Threshold	Number	The minimum amount required

Property	Type	Purpose
		to trigger a hauler dispatch.

By using this interface, the LogisticsNetwork can treat a request from a dropped pile of 1,000 energy and a request from a Tower with 10 energy as comparable data points, allowing for a unified mathematical approach to resource distribution.

Algorithmic Orchestration: Stable Matching with Gale-Shapley

The core of a top-tier logistics broker is the matching algorithm. While a simple greedy algorithm (matching the closest hauler to the highest priority request) is an improvement over legacy roles, it still fails to reach global optimality. The industry standard for high-performance bots is a variation of the **Gale-Shapley algorithm**, also known as the "Stable Marriage" algorithm.

Mathematical Foundations of Stable Matching

The Gale-Shapley algorithm finds a stable matching between two sets of elements (Haulers and Requests) given an ordering of preferences for each element. In the context of Screeps logistics, stability is defined as a state where no hauler and no request mutually prefer each other over their current assignment. If such a mutual preference existed, the hauler would "abandon" its current task to pursue the better one, leading to inefficiency.
The algorithm follows a "Propose and Reject" mechanism:

1. **Proposal:** Each "free" hauler (the proposer) proposes to its most preferred request that it has not yet approached.
2. **Tentative Matching:** Each request (the receiver) maintains its current "best" proposal. If it receives a proposal from a hauler it prefers over its current match, it "rejects" the old hauler and tentatively matches with the new one.
3. **Iteration:** Rejected haulers return to their preference list and propose to their next choice.
4. **Convergence:** The algorithm terminates when all haulers are matched or have exhausted their preference lists.

Heuristics for Preference Ranking

The success of the Gale-Shapley implementation depends on the heuristic used to generate preference lists. A typical ranking function for a hauler might look like:
Where:

- **Priority:** The urgency of the request (e.g., Towers = 10, Extensions = 5, Upgrading = 1).
- **Distance:** The pathfinding distance from the hauler's current position to the target.
- **ResourceDensity:** The ratio of the requested amount to the hauler's capacity, favoring targets that can fully utilize the hauler's move parts.

Because Gale-Shapley is "Proposer-Optimal," if the haulers are the ones proposing, every hauler will receive the best possible target according to its preference list, maximizing the total throughput of the colony.

The Energy Reservation System: Managing "Virtual"

Inventory

Matching is only half of the solution. To prevent "Energy Racing," the broker must implement a robust **Reservation System**. This system tracks resources that are "in flight" but have not yet arrived at their destination.

The Effective Store Calculation

A top-tier broker does not look at the structure.store property in isolation. Instead, it calculates the **Effective Store** :

- **Incoming Reservations:** The sum of energy currently being carried toward this structure by matched haulers.
- **Outgoing Reservations:** The sum of energy that has been promised to haulers currently moving toward this structure to pick up resources.

When the broker matches a hauler to a requester, it immediately adds a reservation to the ledger. If a Tower needs 200 energy and a hauler is dispatched with 200 energy, the broker's ledger will show the Tower's "Effective Store" as full, even though it will take the hauler 10 ticks to arrive. Any subsequent matching logic will see that the Tower's needs are met and will not dispatch another hauler.

Predictive Supply Chain Management

Advanced brokers extend this logic to account for future production. For instance, a static miner at a source produces 10 energy per tick. If a container currently has 500 energy, but a hauler is 20 ticks away, the broker can predict that the container will have 700 energy upon arrival. This "Predictive Amount" allows the broker to dispatch larger haulers or coordinate multiple pickups in a single trip, significantly reducing idle CPU time.

Reservation Type	Definition	Effect on Matching
Pickup Reservation	Hauler is moving to a Provider.	Reduces the "available" energy for other haulers.
Delivery Reservation	Hauler is moving to a Requester.	Increases the "effective" energy, preventing over-filling.
Production Prediction	Energy expected from Sources.	Dispatches haulers to "empty" containers that will be full soon.
Decay Prediction	Expected energy loss from dropped piles.	Prioritizes pickups that are about to vanish.

Hierarchical Coordination: The Overlord and Overseer Pattern

Modern bots like Overmind do not treat logistics as a standalone module but as a service utilized by high-level managers called **Overlords**. This hierarchy ensures that logistics are always aligned with the strategic goals of the empire.

The Colony and Hive Cluster Structure

A "Colony" represents the total presence in a room and its surrounding outposts. Within the

colony, specialized "Hive Clusters" manage specific areas (e.g., the Hatchery for spawning, the Command Center for storage). Each cluster identifies its own logistics needs and registers them with the LogisticsNetwork.
The TransportOverlord (or HaulingOverlord) is responsible for spawning and managing the hauler fleet. It queries the LogisticsNetwork every tick for the list of outstanding matches. If the number of requests exceeds the capacity of the current fleet, the TransportOverlord sends a request to the SpawnHatchery to produce more haulers.

Phase-Based Execution

To maintain stability and avoid race conditions, a top-tier broker executes in distinct phases within the tick.

1. **Build Phase:** Instantiate or refresh persistent classes from the Global Heap.
2. *Registry Phase:* All game objects (Towers, Spawns, Miners) register their TransportRequests.
3. **Matching Phase:** The broker runs the Gale-Shapley algorithm to pair free haulers with the most optimal requests.
4. **Run Phase:** Haulers execute their assigned tasks (moveTo, withdraw, transfer).
5. **Finalize Phase:** Persistent data is updated in the Global Cache, and critical state is saved to Memory.

This structured flow ensures that all "needs" are known before any "actions" are taken, allowing the broker to make decisions with a global view of the room's economy.

Navigation and Cartography: The Logistics of Movement

Movement is the single most CPU-intensive subsystem in Screeps. A hauler that recalculates its path every tick will consume more CPU than the rest of the bot combined.

Path Caching and Serialized Travel

A top-tier broker utilizes a "Path Cache" stored in the Global Heap. When a hauler is assigned a match, the broker calculates the path once using PathFinder.search(). This path is then compressed into a string (e.g., "33214" representing directions) and stored in the hauler's heap memory. On subsequent ticks, the hauler simply reads the next character from the string and moves in that direction, avoiding the O(N) overhead of the built-in pathfinder.

Traffic Management and "Shoving"

In a high-density base, haulers frequently encounter other creeps blocking their path. Standard moveTo() logic causes the creep to wait or recalculate. A modern logistics system implements a "Shove" algorithm. If a high-priority hauler (e.g., carrying energy to a Tower during an attack) is blocked by a lower-priority creep (e.g., an Upgrader), the hauler will "shove" the Upgrader to an adjacent tile. This ensures that the supply chain is never physically interrupted by civilian traffic.

Link Balancing and Instant Transfer

At RCL 5 and above, Links provide the ability to move energy instantly across a room. A modern broker treats Links as a "Short Circuit" in the logistics network. A LinkNetwork process runs prior to the matching phase, identifying "Source Links" (near energy sources) and "Hub Links" (near Storage). If a Source Link is full, it automatically transfers its energy to the Hub Link, effectively removing that energy from the hauler's workload and saving dozens of CPU cycles per tick.

Memory Engineering: The Heap-First Architecture

To survive the "Serialization Wall," top-tier bots move away from Memory as their primary data store. Instead, they use a **Global Cache** pattern.

The Hydration/Dehydration Mechanism

On the first tick of a new isolate (a Global Reset), the bot performs a "Hydration" step. It parses the necessary parts of the Memory object and instantiates rich JavaScript classes for the LogisticsNetwork, Colonies, and Overlords. For the next several thousand ticks, these objects live in the Heap. Accessing global.logisticsNetwork is nanoseconds fast, whereas accessing Memory.logistics requires a proxy-wrapped JSON lookup.

Segmented Memory for Historical Data

For logistics data that is too large for the 2MB Memory limit—such as 10,000 ticks of market price history or massive cost matrices for remote mining—modern bots utilize RawMemory.segments. These 100kb chunks of storage are loaded asynchronously. A top-tier broker uses a "Segment Manager" to request these chunks as needed, allowing the bot to make logistics decisions based on historical trends without bloating the main memory string.

Military Logistics: Defense and War

A logistics broker proves its worth during a siege. In a legacy bot, Towers often run out of energy because haulers are busy filling Extensions or because the hauler was killed and no replacement was dispatched.

Tower Defense Logic and Energy Prioritization

A top-tier broker treats Towers as "Critical Priority" requesters. The moment a Tower fires, its energy level drops, and the LogisticsNetwork registers a new request. Because the Gale-Shapley algorithm uses priority as its primary weight, haulers will immediately drop their current tasks—even if they were moving energy to the Spawn—to refill the Towers. Furthermore, the broker implements "Pre-spawning" for logistics creeps during war. If the Overseer detects a hostile presence, it increases the target hauler count by 20% to account for potential losses and ensures that every hauler is equipped with "Tough" or "Heal" parts to survive moving through a sieged base.

Squad-Based Logistics

For offensive operations, logistics must move with the army. A "Squad" object (often a "Quad" of 4 creeps) includes a dedicated "Healer/Buffer" that functions as a mobile requester. The Logistics Broker coordinates a "Supply Line" of haulers that move energy from the home colony to the front line, potentially spanning multiple rooms. This requires the broker to calculate "Safe Paths" that avoid enemy towers, utilizing the CostMatrix modifications discussed in the Cartography section.

Implementation Roadmap for Modernization

Transforming a legacy codebase (like grgisme/screeps) into a top-tier system requires a phased approach. The goal is to move from a collection of scripts to a managed system of processes.

Phase 1: Toolchain and Environment (The Foundation)

Before any logic is rewritten, the development environment must be upgraded. This involves transitioning from raw JavaScript to TypeScript and implementing a bundling pipeline using Rollup. This allows for the use of complex interfaces and classes, which are essential for the TransportRequest and LogisticsNetwork structures.

Phase 2: The Registry and Kernel (The Core)

The next step is to implement a Kernel that manages the tick lifecycle. The legacy monolithic loop is replaced with a process-based scheduler. Simultaneously, the LogisticsNetwork registry is created. During this phase, creeps still use legacy roles, but they now query the LogisticsNetwork for their targets instead of using room.find().

Phase 3: Stable Matching and Reservations (The Brain)

Once the registry is stable, the "Decision Logic" is moved from the creeps to the broker. The Gale-Shapley algorithm is implemented to assign tasks. The Zerg (creep wrapper) class is introduced to handle the execution of these assigned tasks. The reservation system is activated to eliminate "Energy Racing."

Phase 4: Global Cache and Optimization (The Performance)

The final phase involves moving all persistent state to the Global Heap. The refresh() pattern is implemented to allow class instances to survive between ticks. The path cache and link balancing logic are integrated, resulting in a system that can handle hundreds of creeps with minimal CPU overhead.

Phase	Core Objective	Key Technology	Impact
Phase 1	Modern Tooling	TypeScript, Rollup, ESLint.	Refactoring safety and code modularity.
Phase 2	Centralized Registry	Provider/Requester Registry.	Eliminates O(N) sensing costs.
Phase 3	Optimal Coordination	Gale-Shapley Stable Matching.	Zero energy racing and 100% throughput.
Phase 4	Performance Scaling	Global Heap Cache,	>80% reduction in

Phase	Core Objective	Key Technology	Impact
		Path Caching.	per-tick CPU cost.

Strategic Implications and Future Outlook

The implementation of a Global Logistics Broker represents the "Industrial Revolution" of a Screeps empire. It shifts the player's focus from micro-managing individual units to designing high-level economic and military strategies.

Cross-Shard Economies

As an empire grows beyond GCL 20, the logistics broker must eventually handle inter-shard trade. This involves using InterShardMemory to coordinate the transfer of resources through portals. A top-tier broker will treat a portal as a "Virtual Sink" on one shard and a "Virtual Source" on another, allowing for the seamless flow of energy from a high-yield room on Shard 0 to a fledgling colony on Shard 3.

Market Integration and Automated Trading

A modern logistics broker is also the engine of the bot's economy. Excess energy identified by the broker is automatically sent to the Terminal and sold on the global market. The broker uses market data (stored in memory segments) to determine when to buy minerals for laboratory reactions or when to sell energy for credits. This automated wealth generation provides the resources necessary to sustain high-level warfare and empire expansion.

Conclusion: The Path to Industrial Autonomy

The Screeps architecture landscape has evolved far beyond the simple role-based scripts of the 2017 era. To compete in the 2025 ecosystem, a bot must be treated as a complex systems engineering project. The Global Logistics Broker is the heart of this system, transforming a collection of autonomous agents into a unified, high-throughput supply chain.
By implementing stable matching algorithms to ensure optimal coordination, reservation systems to eliminate resource contention, and a heap-first architecture to bypass serialization bottlenecks, a developer can achieve levels of efficiency that were previously impossible. This architectural shift is not merely an optimization; it is the only viable path to managing the non-linear complexity of a high-GCL empire. The move from "Search and Rescue" to "Command and Control" is the defining characteristic of a top-tier Screeps bot, enabling the player to dominate the world through computational superiority and strategic orchestration.

Works cited

1. Server-side architecture overview | Screeps Documentation, https://docs.screeps.com/architecture.html 2. IVM heap usage & game objects | Screeps Forum, https://screeps.com/forum/topic/2163/ivm-heap-usage-game-objects 3. Screeps #6: Verifiably refreshed - Ben Bartlett, https://bencbartlett.com/blog/screeps-6-verifiably-refreshed/ 4. Global Objects | Screeps Documentation, https://docs.screeps.com/global-objects.html 5. bonzaiferroni/bonzAI-framework - GitHub, https://github.com/bonzaiferroni/bonzAI-framework 6. Releases · bencbartlett/Overmind - GitHub, https://github.com/bencbartlett/Overmind/releases

7. Overmind - Ben Bartlett, https://bencbartlett.com/overmind-docs/ 8. Energy - Screeps Wiki, https://wiki.screepspl.us/Energy/ 9. Issue storing energy in containers :: Screeps: World Help - Steam Community, https://steamcommunity.com/app/464350/discussions/5/3335371283878126074/ 10. Gale–Shapley algorithm - Wikipedia, https://en.wikipedia.org/wiki/Gale%E2%80%93Shapley_algorithm 11. The stable matching algorithm - Washington, https://courses.cs.washington.edu/courses/cse421/25sp/lectures/pdf/lecture2.pdf 12. Implementation of the Gale-Shapley (Stable Matching) algorithm. - GitHub, https://github.com/colorstackorg/stable-matching 13. Gale Shapley Algorithm for Stable Matching | by Sai Panyam | saipanyam - Medium, https://medium.com/saipanyam/gale-shapley-algorithm-for-stable-matching-736ff452dac4 14. Community Communication - Screeps Wiki, https://wiki.screepspl.us/Community_Communication/