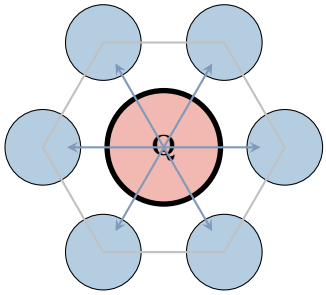


NanOS

Nano Operating System for Disposable Nodes

Manual Técnico

Versión 0.7

Incluye:

Arquitectura del Sistema
Protocolo de Feromonas
NERT: Ephemeral Reliable Transport
Enrutamiento Hebbiano
Stigmergia: Feromonas Digitales
Black Box Distribuida
Hive Bridge: Integración con micrOS
Sintonización Genética (v0.7)
Nodos Judas: Honeypots Activos (v0.7)
Canales Encubiertos (v0.7)
Guía de Implementación

NanOS Project

NanOS Technical Manual
Version 0.5

"Los nodos mueren, el enjambre vive."

Índice general

Índice de figuras

Índice de cuadros

Parte I

NanOS Core

Capítulo 1

Introducción

1.1. ¿Qué es NanOS?

NanOS es un sistema operativo minimalista diseñado para **nodos desechables** en entornos de computación distribuida. A diferencia de los sistemas operativos tradicionales que buscan estabilidad y persistencia, NanOS abraza la **efímera naturaleza** de sus nodos.

Filosofía de Diseño

- **Sin persistencia**: No hay disco, no hay filesystem. Todo es RAM volátil.
- **Ciclo de vida limitado**: Los nodos tienen muerte programada (apoptosis).
- **Regeneración**: Al morir, renacen con nueva identidad.
- **Inteligencia colectiva**: El comportamiento emerge del enjambre.

1.2. Inspiración Biológica

NanOS se inspira en sistemas biológicos como colonias de hormigas y organismos unicelulares. Cada nodo es análogo a una célula que:

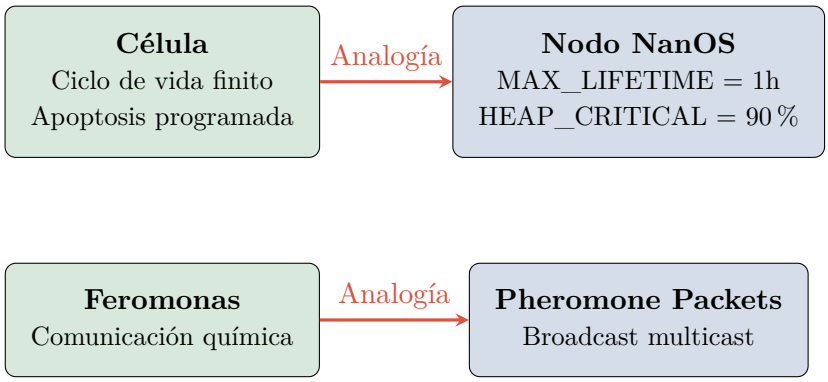


Figura 1.1: Analogía biológica de NanOS

1.3. Plataformas Soportadas

Plataforma	**Arquitectura**	**RAM Típica**	**Uso**
QEMU x86	i386/i686	128KB–1MB	Desarrollo/Testing
ARM Cortex-M3	ARMv7-M	64KB	IoT/Embedded
ESP32	Xtensa LX6	320KB	WiFi/Low-power
ARM64	ARMv8-A	1MB+	Servidores Edge

Cuadro 1.1: Plataformas soportadas por NanOS

Capítulo 2

Arquitectura del Sistema

2.1. Estructura de Capas

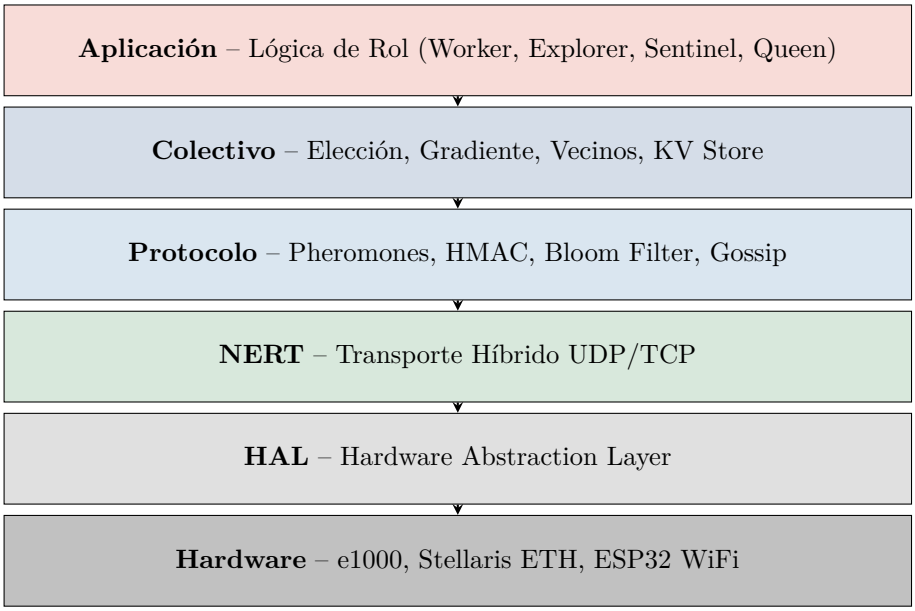


Figura 2.1: Arquitectura de capas de NanOS

2.2. Componentes del Kernel

2.2.1. Gestión de Memoria

NanOS utiliza un heap simple sin fragmentación externa mediante allocación de bloques de tamaño fijo.

```
#define HEAP_SIZE           0x10000     /* 64KB heap */
#define HEAP_CRITICAL_PCT   90          /* Muerte si > 90% */
#define BLOCK_SIZE          64          /* Bloques de 64 bytes */
```

Listing 2.1: Constantes de memoria

2.2.2. Timer y Scheduling

El kernel utiliza un timer de sistema (PIT en x86, SysTick en ARM) para:

- Mantener el contador de ticks global
- Verificar condiciones de apoptosis
- Rotar ventanas del Bloom filter
- Actualizar timeouts de conexiones

2.3. Ciclo de Vida del Nodo

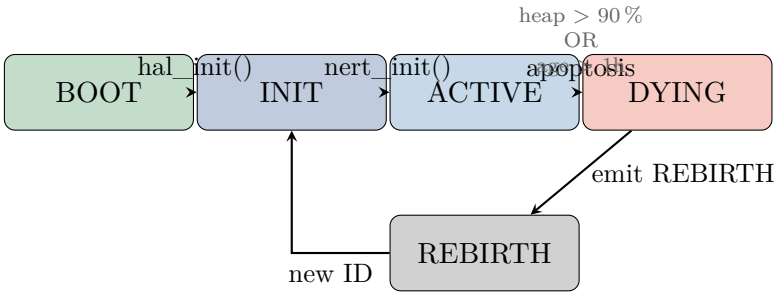


Figura 2.2: Ciclo de vida de un nodo NanOS

2.3.1. Condiciones de Apoptosis

Un nodo entra en estado DYING cuando:

1. **Memoria crítica**: Uso de heap $> 90\,\%$
2. **Antigüedad**: Tiempo de vida > `MAX_CELL_LIFETIME` (3600s)
3. **Comando externo**: Recibe `PHEROMONE_DIE` autenticado

Capítulo 3

Sistema de Roles

3.1. Roles Definidos

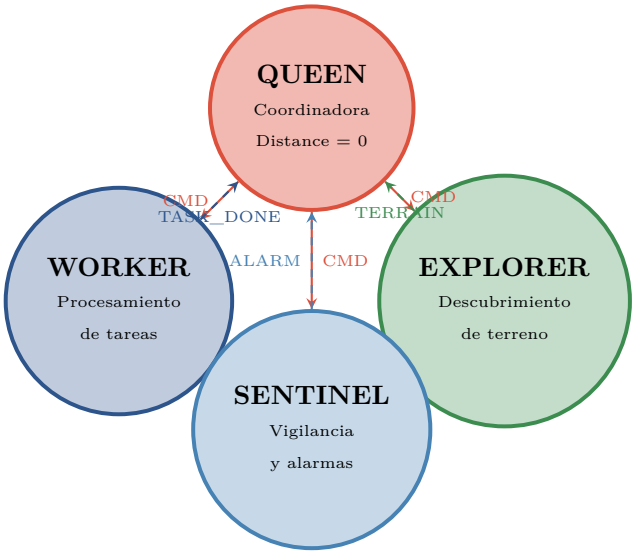


Figura 3.1: Sistema de roles en el enjambre

3.2. Transición de Roles

Los nodos pueden cambiar de rol dinámicamente basándose en:

- **Necesidades del enjambre**: Si hay déficit de exploradores, workers pueden transicionar
- **Muerte de la reina**: Se inicia proceso de elección
- **Comando de la reina**: Reasignación directa

Condición	Umbral	Acción
Sentinelas $< 10\,\%$	MIN_SENTINEL_RATIO	Worker $\rightarrow$ Sentinel
Exploradores $< 10\,\%$	MIN_EXPLORER_RATIO	Worker $\rightarrow$ Explorer
Sin reina detectada	QUEEN_TIMEOUT	Iniciar elección

Cuadro 3.1: Reglas de transición de roles

3.3. Algoritmo de Elección de Reina

El algoritmo de elección es **determinístico** y garantiza convergencia:

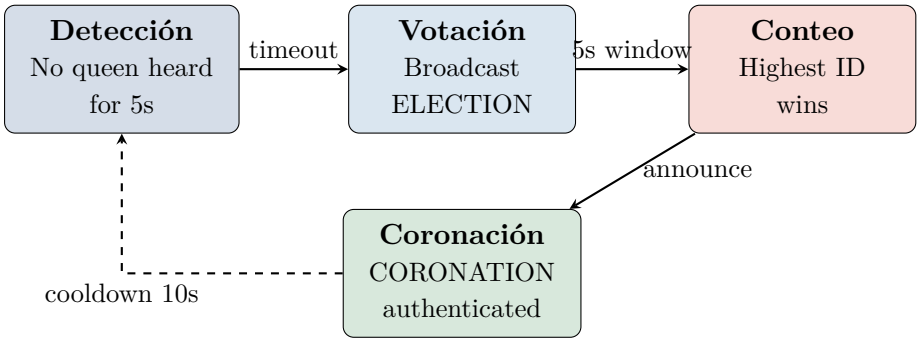


Figura 3.2: Fases del algoritmo de elección

```c
struct election_state {
    uint32_t election_id;       /* ID unico de eleccion */
    uint32_t started_at;        /* Tick de inicio */
    uint32_t my_vote;           /* A quien voto */
    uint32_t highest_vote_id;   /* Maximo ID visto */
    uint8_t  participating;     /* Participando? */
    uint8_t  phase;             /* 0=none, 1=voting, 2=counting */
};
```

Listing 3.1: Estructura de estado de elección

Capítulo 4

Protocolo de Feromonas

4.1. Tipos de Pheromones

Código	Nombre	Auth	Descripción
0x01	HELLO	No	Heartbeat con información de gradiente
0x02	DATA	No	Transporte de datos genérico
0x03	ALARM	No	Alerta de peligro detectado
0x04	ECHO	No	Respuesta/reconocimiento
0x05	ELECTION	No	Voto en elección de reina
0x06	CORONATION	**Sí**	Anuncio de nueva reina
0x10	QUEEN_CMD	**Sí**	Comando de la reina
0x20–0x22	KV_*	No	Key-Value store distribuido
0x30	TASK	No	Asignación de tarea
0x40	SENSOR	No	Lectura de sensor
0x70–0x73	MAZE_*	No	Exploración de laberinto
0x80–0x87	TERRAIN_*	No	Mapeo de terreno
0xFE	REBIRTH	**Sí**	Notificación de muerte/renacimiento
0xFF	DIE	**Sí**	Comando de terminación
Extensiones v0.7			
0x14	CONFIG_UPDATE	**Sí**	Actualización de genoma (Genetic Tuning)
0x15	TELEMETRY_REPORT	No	Reporte de telemetría a Queen
0x16	JUDAS_ENGAGE	No	Notifica enganche de atacante
0x17	JUDAS_CAPTURE	No	Payload de atacante capturado
0x18	JUDAS_FORENSICS	No	BlackBox forense antes de apoptosis

Cuadro 4.1: Tipos de pheromones y sus características

4.2. Formato de Paquete

4.2.1. Paquete Estándar (64 bytes)

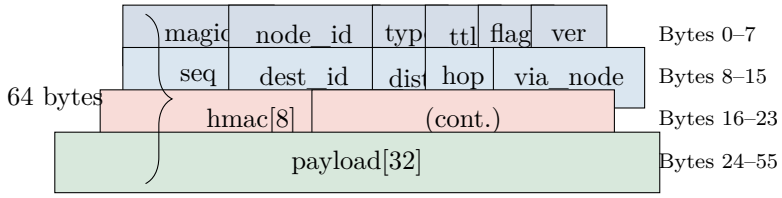


Figura 4.1: Estructura del paquete pheromone estándar

4.2.2. Paquete Compacto ARM (24 bytes)

Para plataformas con recursos limitados, se usa un formato compacto con **62 % menos overhead**:

```
struct nanos_pheromone_compact {
    uint8_t  magic;          /* 0xAA */
    uint16_t node_id;        /* 16-bit truncado */
    uint8_t  type;
    uint8_t  ttl_flags;      /* TTL(4) + flags(4) */
    uint8_t  seq;            /* 8-bit sequence */
    uint16_t dest_id;
    uint8_t  dist_hop;       /* distance(4) + hop(4) */
    uint8_t  payload[8];
    uint8_t  hmac[4];        /* 4-byte truncado */
    uint8_t  reserved[3];
};  /* Total: 24 bytes */
```

Listing 4.1: Estructura del paquete compacto

4.3. Autenticación HMAC

Los paquetes críticos requieren autenticación mediante un HMAC basado en SipHash simplificado:

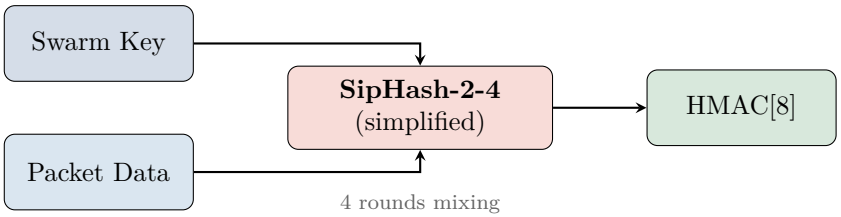


Figura 4.2: Proceso de generación de HMAC

Capítulo 5

Mecanismos de Red

5.1. Bloom Filter para Deduplicación

El Bloom filter proporciona deduplicación $O(1)$ con uso mínimo de memoria:

```
#define BLOOM_BITS       256      /* 32 bytes */
#define BLOOM_HASH_K     3        /* 3 funciones hash */
#define BLOOM_SLOTS      4        /* Ventanas rotativas */
#define BLOOM_WINDOW_MS  500      /* 500ms por ventana */
```

Listing 5.1: Configuración del Bloom filter

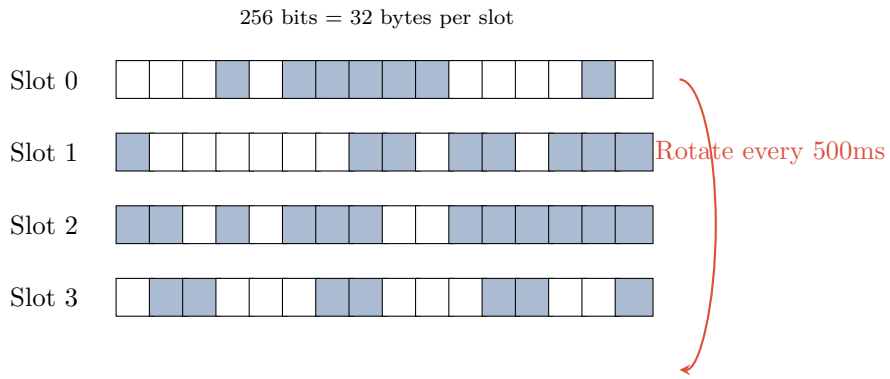


Figura 5.1: Visualización del Bloom filter con slots rotativos

5.2. Gossip Protocol

El protocolo de gossip controla la propagación de mensajes para evitar tormentas de broadcast:

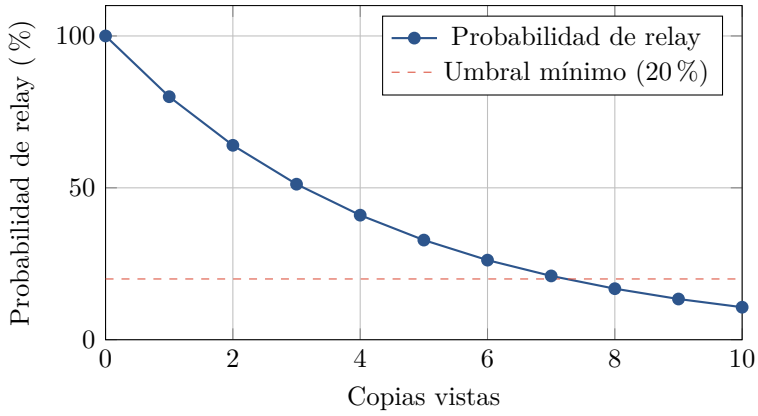


Figura 5.2: Decaimiento de probabilidad en el protocolo gossip

5.3. Gradient Routing

El enrutamiento por gradiente permite dirigir mensajes hacia la reina:

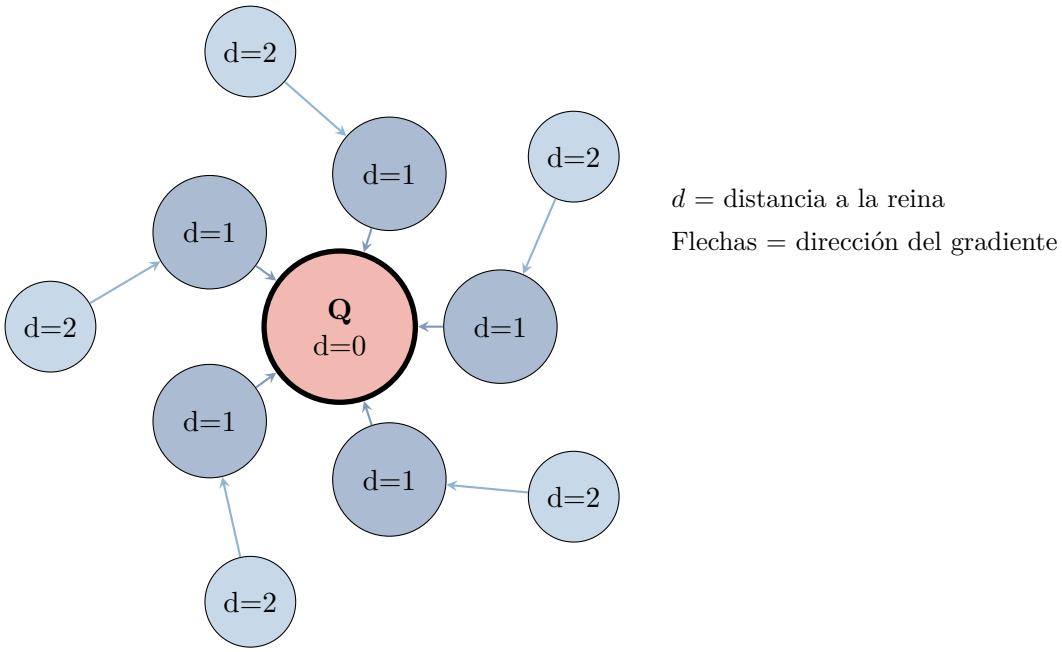


Figura 5.3: Enrutamiento por gradiente hacia la reina

5.4. Enrutamiento Hebbiano (v0.5)

La versión 0.5 introduce **Enrutamiento Hebbiano**, inspirado en la neurociencia: *"Las neuronas que se disparan juntas, se conectan juntas"*.

5.4.1. Peso Sináptico

Cada conexión con un vecino tiene un **peso sináptico** $w \in [1, 255]$:

```
struct neighbor_entry {
    uint32_t node_id;
    uint32_t last_seen;
    uint8_t  role;
```

```
    uint8_t  distance;
    uint16_t packets;
    uint8_t  synaptic_weight;  /* v0.5: 0-255, inicial: 128 */
};
```

Listing 5.2: Campos de peso sináptico

5.4.2. Regla de Aprendizaje

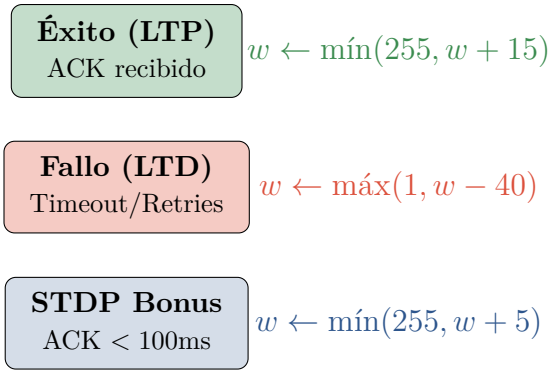


Figura 5.4: Reglas de aprendizaje Hebbiano

Asimetría Castigo/Recompensa

El castigo (-40) es casi 3 veces más severo que la recompensa ($+15$). Esto asegura que el enjambre aprenda rápidamente a evitar nodos poco confiables.

5.4.3. Fórmula de Costo Neural

El costo de ruta combina distancia y confiabilidad:

$$\text{Costo}(j) = 10 \cdot d_j + \frac{255 - w_j}{8} \tag{5.1}$$

Distancia	Peso	Costo
1 hop	255 (perfecto)	$10 + 0 = 10$
1 hop	128 (neutral)	$10 + 15 = 25$
1 hop	1 (muerto)	$10 + 31 = 41$
2 hops	255 (perfecto)	$20 + 0 = 20$

Cuadro 5.1: Ejemplos de cálculo de costo neural

Insight Clave

Una ruta de 2 saltos confiable (costo 20) es mejor que una ruta de 1 salto poco confiable (costo 41). El enjambre **aprende a rodear** nodos problemáticos.

Capítulo 6

Stigmergia: Feromonas Digitales (v0.5)

6.1. Inspiración Biológica

Las hormigas no memorizan mapas; dejan **químicos que se evaporan**. Este mecanismo de coordinación indirecta a través de modificación del ambiente se llama **stigmergia**. NanOS v0.5 implementa este concepto digitalmente.

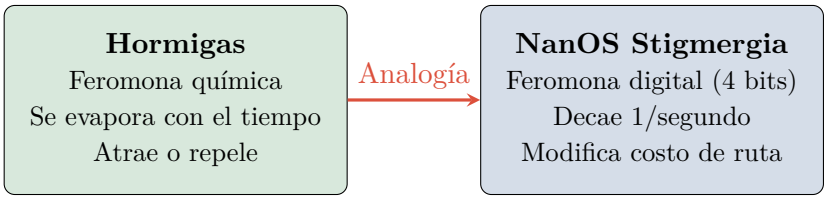


Figura 6.1: Analogía biológica de Stigmergia

6.2. Tipos de Feromonas

Código	Tipo	Efecto	Uso
0	DANGER	Repulsión (+8/nivel)	Jamming, ataques, nodos maliciosos
1	QUEEN	Atracción (-2/nivel)	Camino hacia la reina
2	RESOURCE	Neutral	Marcador de objetivos
3	AVOID	Repulsión (+4/nivel)	Zonas subóptimas (no peligrosas)

Cuadro 6.1: Tipos de feromonas digitales

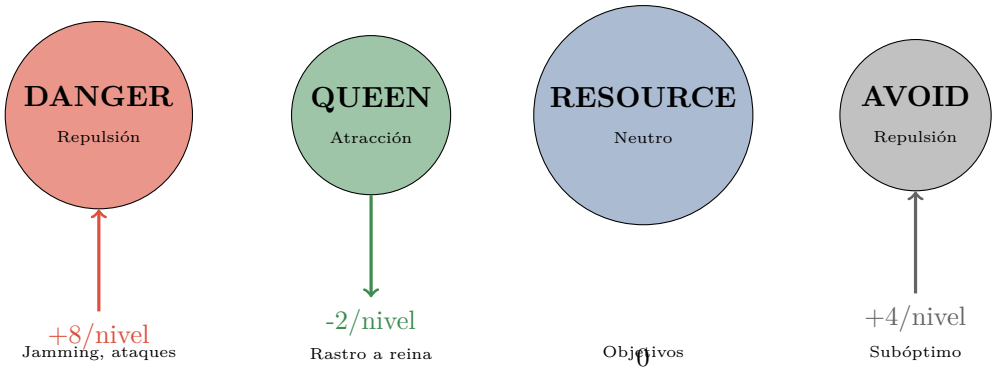


Figura 6.2: Tipos de feromonas y su efecto en el costo de movimiento

6.3. Estructura de Datos

```c
/* Cada celda almacena 4 tipos en 2 bytes (nibble packing) */
struct stigmergia_cell {
    uint8_t data[2];  /* [danger|queen], [resource|avoid] */
};

/* Grid 16x16 cubriendo terreno 32x32 (escala 2:1) */
struct stigmergia_cell pheromones[16][16];  /* 512 bytes total */

/* Acceso a intensidad (0-15) */
uint8_t stigmergia_get(uint8_t x, uint8_t y, uint8_t type);
void stigmergia_mark(uint8_t x, uint8_t y, uint8_t type, uint8_t
    intensity);
```

Listing 6.1: Almacenamiento de feromonas

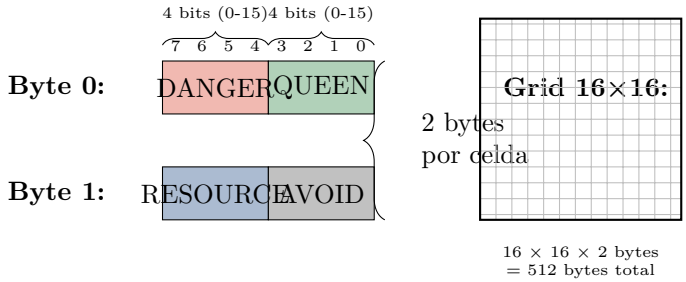


Figura 6.3: Nibble packing: 4 tipos de feromona en 2 bytes por celda

6.4. Decaimiento Temporal

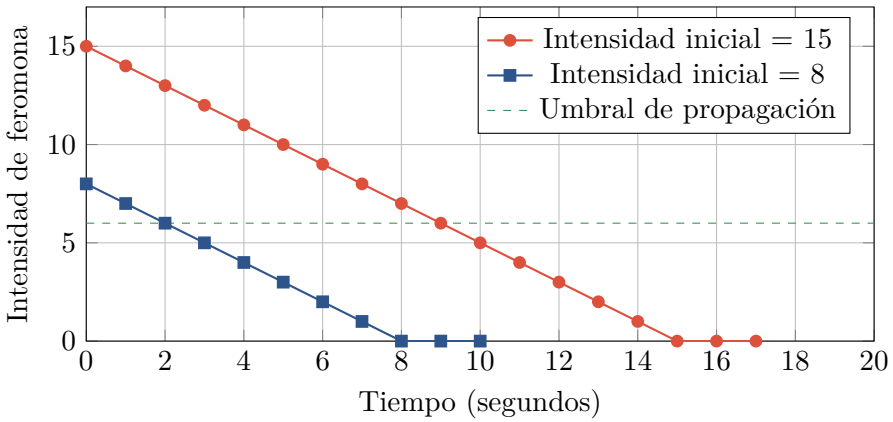


Figura 6.4: Decaimiento de feromonas: -1 cada segundo

Evaporación

La feromona **se pierde** si no es reforzada. Un peligro detectado hace 15 segundos ya no existe en el mapa. Esto evita información obsoleta.

6.5. Modificación de Costo de Movimiento

La presencia de feromonas modifica el costo de moverse a una celda:

$$\text{Costo}_{\text{total}} = \text{Costo}_{\text{base}} + 8 \cdot I_{\text{DANGER}} + 4 \cdot I_{\text{AVOID}} - 2 \cdot I_{\text{QUEEN}} \tag{6.1}$$

Situación	Feromonas	Modificador	Costo Final
Celda normal	Ninguna	0	10
Zona peligrosa	DANGER=15	+120	130
Cerca de reina	QUEEN=10	-20	-10 (atracción)
Zona subóptima	AVOID=8	+32	42
Mixta	DANGER=5, QUEEN=3	+40-6=+34	44

Cuadro 6.2: Ejemplos de modificación de costo

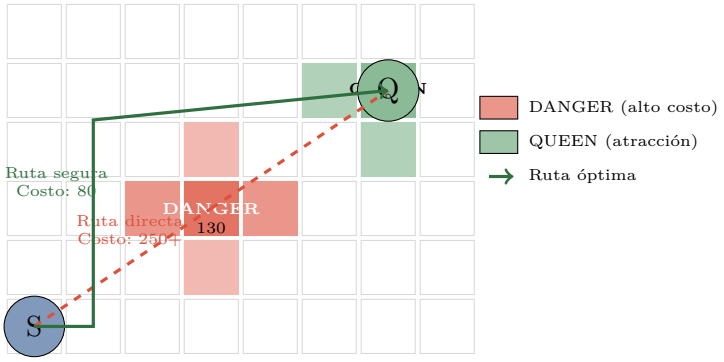


Figura 6.5: Efecto de feromonas en pathfinding: el enjambre rodea zonas peligrosas

6.6. Propagación de Feromonas

Las feromonas de alta intensidad se propagan a celdas vecinas:

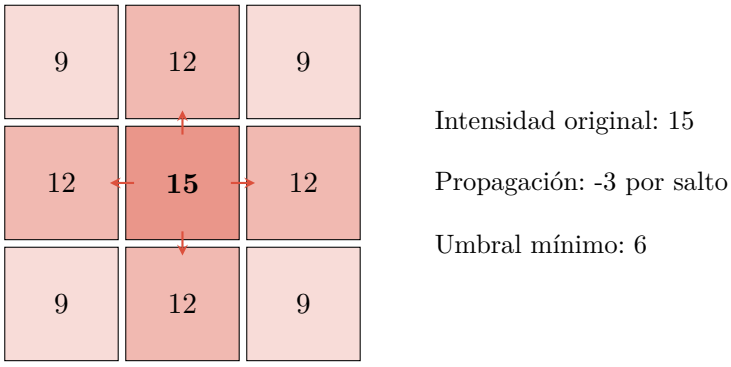


Figura 6.6: Propagación de feromona DANGER

6.7. API de Stigmergia

```c
/* Inicializar sistema (limpia todas las feromonas) */
void stigmergia_init(void);

/* Marcar feromona en coordenadas de terreno */
void stigmergia_mark(uint8_t terrain_x, uint8_t terrain_y,
                     uint8_t type, uint8_t intensity);

/* Obtener intensidad en coordenadas */
uint8_t stigmergia_get(uint8_t terrain_x, uint8_t terrain_y,
                       uint8_t type);

/* Aplicar decaimiento (llamar cada segundo) */
void stigmergia_decay(void);

/* Calcular modificador de costo para pathfinding */
int8_t stigmergia_cost_modifier(uint8_t terrain_x, uint8_t terrain_y);

/* Emitir feromona de peligro (convenience) */
void stigmergia_emit_danger(uint8_t intensity);

/* Emitir rastro hacia la reina */
void stigmergia_emit_queen_trail(void);
```

Listing 6.2: Funciones principales de Stigmergia

Capítulo 7

Black Box Distribuida: "El Último Aliento" (v0.5)

7.1. Problema: Evidencia Forense Perdida

Cuando un nodo es comprometido y terminado, su evidencia forense se pierde. Un atacante inteligente podría:

1. Comprometer un nodo
2. Extraer información sensible
3. Forzar su "suicidio" para eliminar rastros

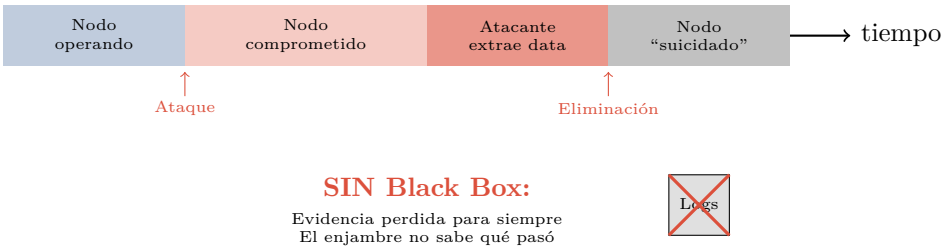


Figura 7.1: Escenario de ataque sin Black Box: la evidencia se pierde al morir el nodo

Escenario de Ataque

Sin Black Box: El atacante elimina el nodo y toda evidencia de compromiso desaparece. El enjambre no sabe qué pasó.

7.2. Solución: El Último Aliento

Antes de morir, cada nodo transmite un "testamento" (Last Will) a vecinos de confianza. Incluso si el nodo fue hackeado y suicidado, su evidencia sobrevive en el enjambre.

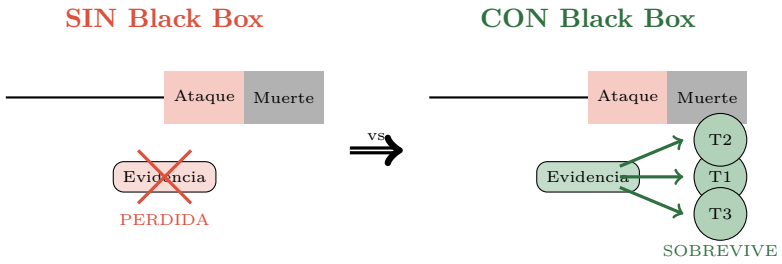


Figura 7.2: Comparación: sin Black Box la evidencia se pierde, con Black Box sobrevive

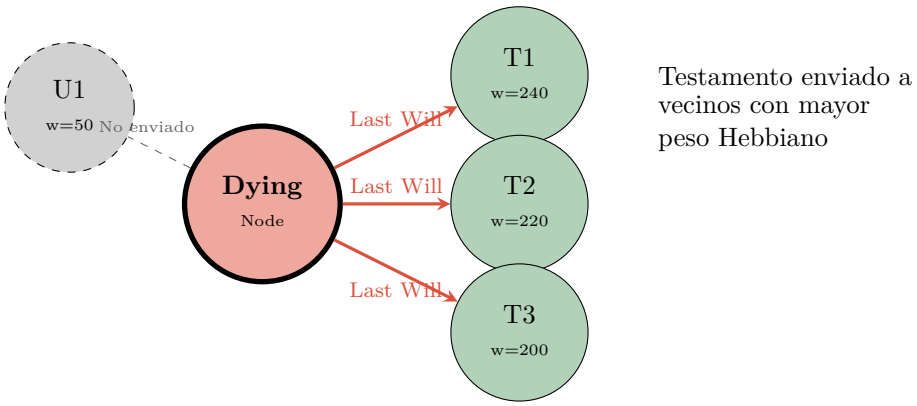


Figura 7.3: Transmisión del Último Aliento a vecinos de confianza

7.3. Contenido del Testamento

```c
/* Razones de muerte */
#define DEATH_NATURAL           0x00  /* Vejez/timeout normal */
#define DEATH_HEAP_EXHAUSTED    0x01  /* Sin memoria */
#define DEATH_CORRUPTION        0x02  /* Corrupcion detectada */
#define DEATH_ATTACK_DETECTED   0x03  /* Ataque en progreso */
#define DEATH_QUEEN_ORDER       0x04  /* Orden de la reina */
#define DEATH_ISOLATION         0x05  /* Sin contacto con enjambre */

/* Tipos de eventos de seguridad */
#define EVENT_BAD_MAC           0x01  /* MAC invalido recibido */
#define EVENT_REPLAY            0x02  /* Intento de replay */
#define EVENT_RATE_LIMIT        0x03  /* Rate limit excedido */
#define EVENT_BLACKLIST         0x04  /* Nodo en blacklist */
#define EVENT_JAMMING           0x05  /* Jamming detectado */
#define EVENT_CORRUPTION        0x06  /* Corrupcion de memoria */

/* Contenido del testamento */
struct last_will {
    uint32_t node_id;           /* ID del nodo que muere */
    uint8_t  death_reason;      /* DEATH_* codigo */
    uint8_t  uptime_hours;      /* Horas de vida */
    uint16_t bad_mac_count;     /* MACs invalidos recibidos */
    uint16_t replay_count;      /* Intentos de replay */
    uint16_t rate_limit_hits;   /* Veces rate limited */

    /* Ultimos 8 eventos de seguridad */
    struct {
```

```
        uint8_t  type;          /* EVENT_* tipo */
        uint16_t source_node;   /* Nodo relacionado */
        uint32_t timestamp;     /* Cuando ocurrio */
    } events[8];
};
```

Listing 7.1: Estructura del Last Will

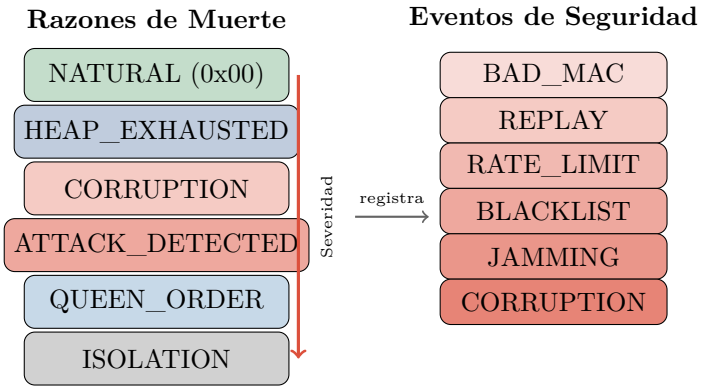


Figura 7.4: Códigos de muerte y eventos de seguridad registrados en el testamento

7.4. Flujo de Operación

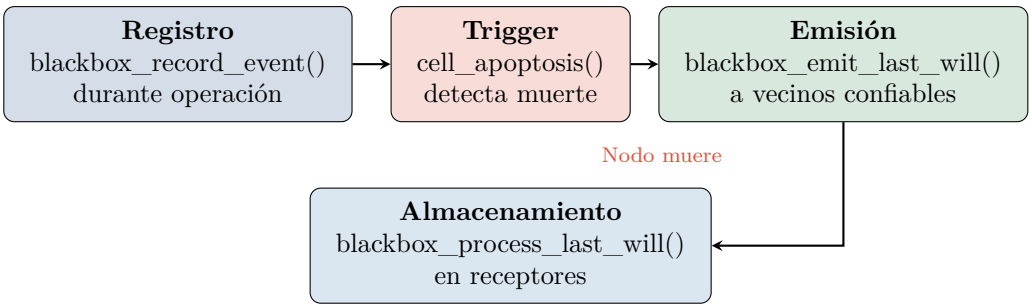


Figura 7.5: Flujo del sistema Black Box

7.5. Almacenamiento de Testamentos

Cada nodo almacena hasta 8 testamentos recibidos:

```
#define BLACKBOX_MAX_WILLS  8

/* Estructura de almacenamiento */
struct blackbox_storage {
    struct {
        uint32_t node_id;
        uint8_t  death_reason;
        uint8_t  uptime_hours;
        uint16_t bad_mac_count;
        uint16_t replay_count;
        uint8_t  priority;      /* Para relay y reemplazo */
        uint8_t  event_count;
    } wills[BLACKBOX_MAX_WILLS];

```

```
    uint8_t count;              /* Testamentos almacenados */
};
```

Listing 7.2: Almacenamiento de testamentos

7.6. API de Black Box

```
/* Inicializar sistema */
void blackbox_init(void);

/* Registrar evento de seguridad (durante operacion) */
void blackbox_record_event(uint8_t event_type, uint16_t source_node);

/* Emitir testamento antes de morir */
void blackbox_emit_last_will(uint8_t death_reason);

/* Procesar testamento recibido */
void blackbox_process_last_will(struct nanos_pheromone* pkt);

/* Consultar muerte de un nodo */
int blackbox_query_death(uint32_t node_id,
                         uint8_t *death_reason,
                         uint16_t *bad_mac_count,
                         uint8_t *uptime_hours);

/* Obtener numero de testamentos almacenados */
uint8_t blackbox_get_will_count(void);

/* Propagar testamentos criticos a vecinos */
void blackbox_relay_critical(void);

/* Imprimir resumen forense (debug) */
void blackbox_print_summary(void);
```

Listing 7.3: Funciones principales de Black Box

7.7. Ejemplo de Investigación Forense

```
void investigate_dead_node(uint32_t suspect_id) {
    uint8_t death_reason;
    uint16_t bad_mac;
    uint8_t uptime;

    if (blackbox_query_death(suspect_id, &death_reason,
                             &bad_mac, &uptime) == 0) {
        printf("Node 0x%08x death investigation:\n", suspect_id);
        printf("  Uptime: %d hours\n", uptime);

        switch (death_reason) {
            case DEATH_ATTACK_DETECTED:
                printf("  ALERT: Died under attack!\n");
                printf("  Bad MACs received: %d\n", bad_mac);
                break;
            case DEATH_CORRUPTION:
                printf("  Memory corruption detected\n");
```

```
                break;
            case DEATH_NATURAL:
                printf("  Normal lifecycle end\n");
                break;
        }
    } else {
        printf("No forensic data for node 0x%08x\n", suspect_id);
    }
}
```

Listing 7.4: Consulta forense de nodo muerto

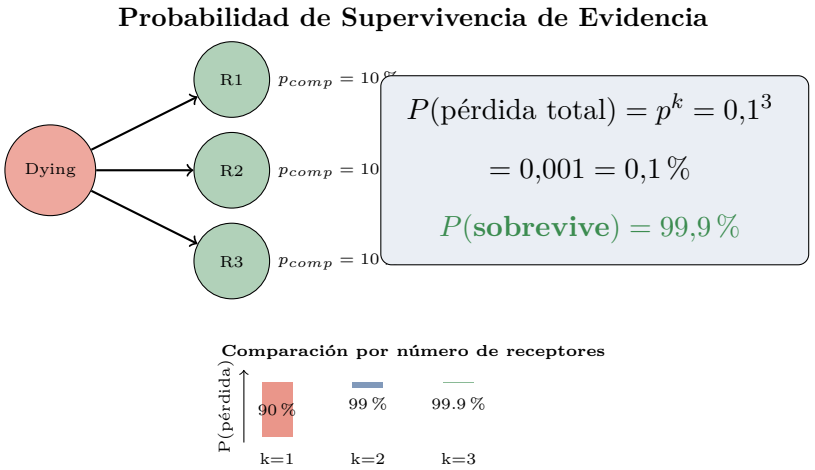


Figura 7.6: Supervivencia de evidencia: con 3 receptores, la probabilidad de pérdida es solo 0.1 %

Supervivencia de Evidencia

Con 3 receptores de confianza y probabilidad de compromiso 10 % por nodo, la evidencia sobrevive con probabilidad $1 - 0{,}1^3 = 99{,}9\,\%$. “Los muertos hablan a través de los vivos.”

Capítulo 8

Sistema Inmune Artificial (AIS) (v0.6)

El sistema inmune biológico no necesita conocer los patógenos de antemano — simplemente reconoce lo que **no es propio** y lo elimina. NanOS v0.6 implementa este concepto con el algoritmo de **Selección Negativa**.

Filosofía del AIS

"El sistema inmune del swarm no necesita saber qué aspecto tienen los ataques — solo necesita saber qué aspecto tiene 'saludable'."

8.1. Conceptos Biológicos

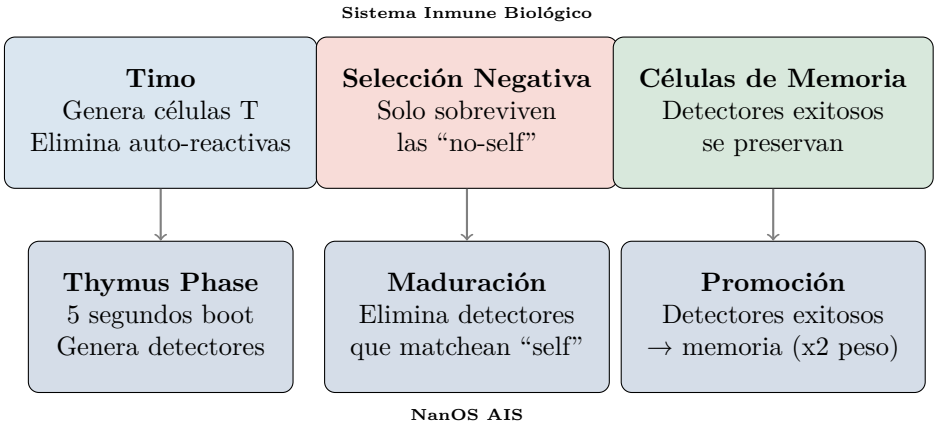


Figura 8.1: Mapeo de conceptos inmunológicos a la implementación NanOS

8.2. Estructura del Detector

Un detector es un "anticuerpo digital" que reconoce patrones anómalos:

```
#define AIS_DETECTOR_COUNT      16      /* Detectores activos */
#define AIS_DETECTOR_SIZE       8       /* Bytes por patron */
#define AIS_AFFINITY_THRESHOLD  6       /* Bits contiguos para match */

struct ais_detector {
    uint8_t  pattern[8];        /* Patron del detector */
    uint8_t  mask[8];           /* Que bits comparar */
    uint8_t  state;             /* EMPTY, IMMATURE, MATURE, MEMORY */
    uint8_t  detect_type;       /* Tipo de deteccion si conocido */
```

```
    uint16_t matches;           /* Veces que matcheo no-self */
    uint16_t false_positives;   /* Matcheo self (malo) */
    uint32_t created_tick;      /* Cuando se creo */
    uint32_t last_match_tick;   /* Ultimo match exitoso */
};

/* Estados del detector */
#define AIS_DETECTOR_EMPTY      0x00    /* Slot disponible */
#define AIS_DETECTOR_IMMATURE   0x01    /* En thymus, aprendiendo */
#define AIS_DETECTOR_MATURE     0x02    /* Activo, detecta no-self */
#define AIS_DETECTOR_MEMORY     0x03    /* Promovido a memoria */
#define AIS_DETECTOR_ANERGIC    0x04    /* Deshabilitado (matcheo self)
   */
```

Listing 8.1: Estructura del detector AIS

8.3. Algoritmo de Selección Negativa

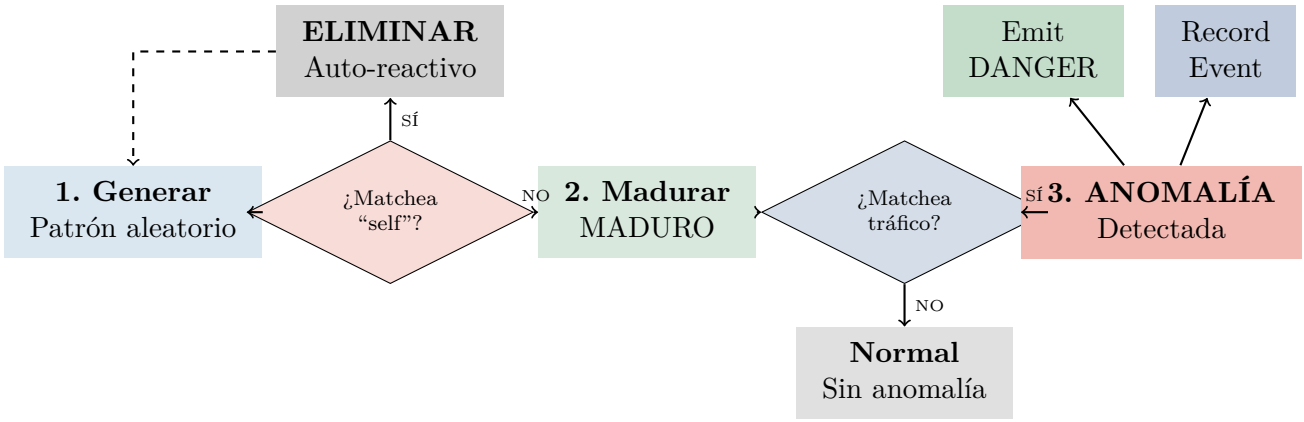


Figura 8.2: Flujo del algoritmo de Selección Negativa

8.4. Extracción de Antígenos

Cada paquete se convierte en un "antígeno" de 8 bytes para comparación:

```
struct ais_antigen {
    uint8_t features[8];        /* Vector de caracteristicas */
    uint16_t source_node;       /* Nodo origen */
    uint8_t pheromone_type;     /* Tipo de paquete */
    uint8_t context_flags;      /* Contexto de seguridad */
};

/* Caracteristicas extraidas del paquete */
void ais_extract_antigen(struct nanos_pheromone* pkt,
                         struct ais_antigen* antigen,
                         uint8_t ctx_flags) {
    antigen->features[0] = pkt->type;           /* Tipo */
    antigen->features[1] = pkt->ttl;            /* TTL */
    antigen->features[2] = pkt->hop_count;      /* Saltos */
    antigen->features[3] = (uint8_t)pkt->node_id;  /* ID origen */
    antigen->features[4] = pkt->distance;       /* Distancia */
    antigen->features[5] = ctx_flags;           /* Contexto */
    antigen->features[6] = (uint8_t)pkt->seq;   /* Secuencia */
```

```
    antigen->features[7] = pkt->flags;          /* Flags */
}
```

Listing 8.2: Extracción de características del paquete

8.5. Tipos de Detección

Código	Tipo	Descripción
0x01	FLOOD	Patrón de DoS/flooding
0x02	PROBE	Reconocimiento/escaneo de red
0x03	REPLAY	Patrón de ataque de replay
0x04	INJECTION	Inyección de paquetes maliciosos
0x05	BEHAVIORAL	Comportamiento anómalo de nodo
0x06	SYBIL	Múltiples identidades desde una fuente
0xFF	UNKNOWN	**Anomalía desconocida (0-day!)**

Cuadro 8.1: Tipos de detección del AIS

8.6. Perfil "Self" (Qué es Normal)

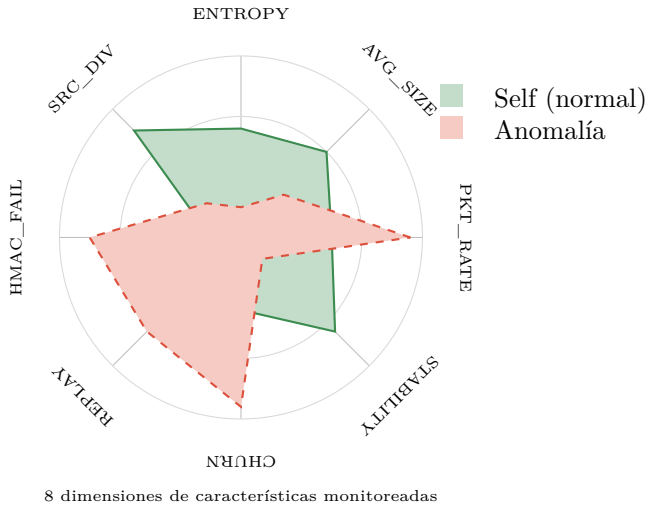


Figura 8.3: Perfil "Self" vs anomalía en el espacio de características

```
/* Dimensiones del perfil self (normalizadas 0-255) */
#define AIS_FEATURE_PKT_RATE        0   /* Paquetes por segundo */
#define AIS_FEATURE_AVG_SIZE        1   /* Tamanio promedio */
#define AIS_FEATURE_TYPE_ENTROPY    2   /* Entropia de tipos */
#define AIS_FEATURE_SRC_DIVERSITY   3   /* Diversidad de fuentes */
#define AIS_FEATURE_HMAC_FAIL_RATE  4   /* Tasa de fallos HMAC */
#define AIS_FEATURE_REPLAY_RATE     5   /* Tasa de replays */
#define AIS_FEATURE_NEIGHBOR_CHURN  6   /* Cambios de vecinos */
#define AIS_FEATURE_ROUTE_STABILITY 7   /* Estabilidad de rutas */
```

Listing 8.3: Características monitoreadas para el perfil "self"

8.7. Integración con Otros Sistemas

Cuando el AIS detecta una anomalía, activa una respuesta coordinada:

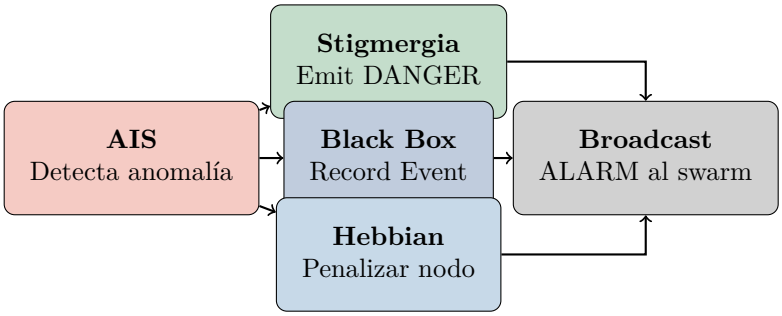


Figura 8.4: Respuesta coordinada del AIS con otros sistemas NanOS

8.8. API del Sistema Inmune

```c
/* Inicializar AIS - inicia thymus phase */
void ais_init(void);

/* Llamar periodicamente desde main loop */
void ais_tick(void);

/* Procesar paquete a traves del sistema inmune */
uint8_t ais_process_packet(struct nanos_pheromone* pkt, uint8_t
    ctx_flags);

/* Verificar si maduracion completa */
bool ais_is_mature(void);

/* Obtener estadisticas */
uint8_t ais_get_active_detector_count(void);
uint32_t ais_get_detection_count(void);

/* Debug */
void ais_print_status(void);
```

Listing 8.4: Funciones principales del AIS

8.9. Eventos AIS para Black Box

```c
#define EVENT_AIS_DETECTOR_MATCH     0x10    /* Detector matcheo no-self
    */
#define EVENT_AIS_THYMUS_COMPLETE    0x11    /* Maduracion completa */
#define EVENT_AIS_MEMORY_PROMOTE     0x12    /* Detector promovido */
#define EVENT_AIS_ANOMALY_ALERT      0x13    /* Umbral de anomalia */
#define EVENT_AIS_SELF_UPDATE        0x14    /* Perfil self actualizado
    */
```

Listing 8.5: Eventos del AIS registrados en Black Box

Detección de 0-Days

El AIS no necesita firmas de ataques conocidos. Cualquier patrón que no coincida con el perfil "self" aprendido durante la fase thymus es considerado sospechoso. Esto permite detectar ataques completamente nuevos (0-days) que nunca se han visto antes.

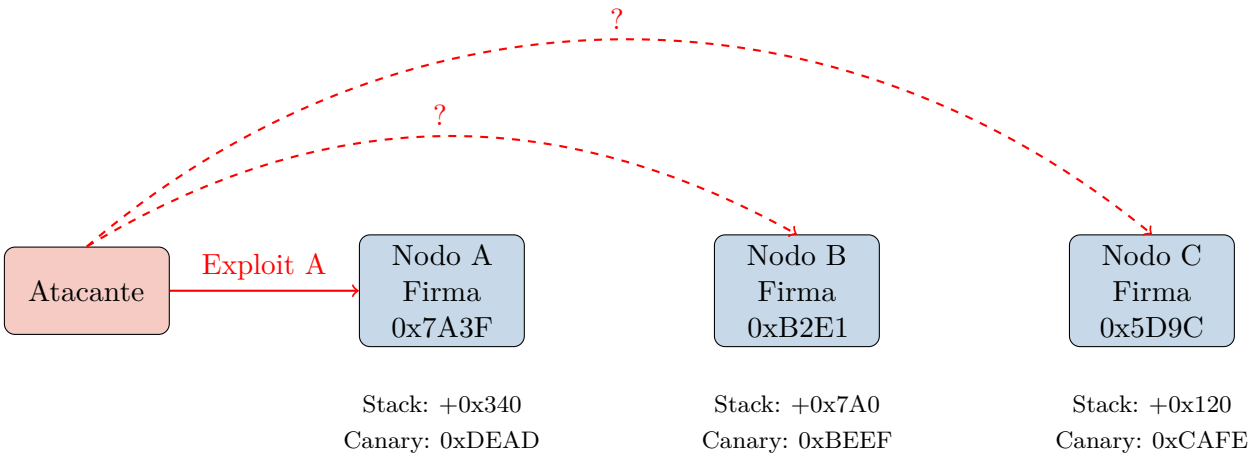
Capítulo 9

Polimorfismo de Código: "El Camaleón" (v0.6)

El módulo de **Polimorfismo de Código** ("El Camaleón") proporciona diversidad a nivel binario, asegurando que cada nodo del swarm tenga una huella digital única. Esto hace imposible reutilizar exploits entre nodos.

9.1. Motivación

En ataques tradicionales, una vez que un atacante desarrolla un exploit para una vulnerabilidad, puede reutilizarlo contra cualquier sistema que ejecute el mismo binario. El polimorfismo rompe este paradigma:



Principio Fundamental

"Si un atacante compromete un nodo, ha ganado una batalla. Para comprometer el swarm, tendría que ganar miles de batallas diferentes."

9.2. Componentes del Polimorfismo

9.2.1. ASLR (Address Space Layout Randomization)

Randomización del layout de memoria al boot:

```c
/* Bits de entropia (limitados por constraints de MCU) */
#define POLY_ASLR_STACK_BITS    8   /* 256 bases posibles para stack */
#define POLY_ASLR_HEAP_BITS     6   /* 64 bases posibles para heap */
#define POLY_ASLR_ALIGNMENT     16  /* Alineacion de 16 bytes (ARM) */

#define POLY_STACK_OFFSET_MAX   4096    /* Max randomizacion stack */
#define POLY_HEAP_OFFSET_MAX    1024    /* Max randomizacion heap */

struct poly_memory_layout {
    uint32_t stack_base;        /* Base randomizada del stack */
    uint32_t stack_offset;      /* Offset desde base original */
    uint32_t heap_base;         /* Base randomizada del heap */
    uint32_t heap_offset;       /* Offset desde base original */
    uint32_t data_offset;       /* Offset de datos estaticos */
    uint32_t entropy_seed;      /* Semilla usada */
};
```

Listing 9.1: Configuración de ASLR

9.2.2. Stack Canaries

Valores aleatorios para detectar buffer overflows:

```c
struct poly_canary_state {
    uint32_t value;         /* Valor actual del canary */
    uint32_t backup;        /* Backup para verificacion */
    uint32_t violations;    /* Contador de violaciones */
    uint32_t last_check;    /* Ultimo tick de verificacion */
    uint32_t last_refresh;  /* Ultimo tick de refresh */
};

/* Macros de instrumentacion */
#define POLY_CANARY_PROLOGUE() \
    uint32_t __canary = poly_canary_get()
#define POLY_CANARY_EPILOGUE() \
    if (__canary != poly_canary_get()) poly_canary_violated()

/* Ejemplo de uso */
POLY_PROTECTED_FUNC void critical_operation(void) {
    POLY_CANARY_PROLOGUE();
    /* ... codigo critico ... */
    POLY_CANARY_EPILOGUE();
}
```

Listing 9.2: Protección con Stack Canary

9.2.3. Firma Binaria Única

Cada nodo genera una firma de 128 bits que lo identifica:

```c
#define POLY_SIGNATURE_SIZE     16  /* 128 bits */

struct poly_signature {
    uint8_t  bytes[POLY_SIGNATURE_SIZE];  /* Fingerprint unico */
    uint32_t created_tick;                /* Cuando fue generado */
    uint32_t node_id;                     /* ID del nodo asociado */
    uint8_t  version;                     /* Version de firma */
```

```
};

/* La firma se deriva de:
 * - Node ID
 * - Hardware RNG seed
 * - Boot timestamp
 * - Memory layout entropy
 */
```

Listing 9.3: Estructura de Firma Binaria

9.2.4. Timing Jitter

Delays aleatorios para resistir ataques de timing side-channel:

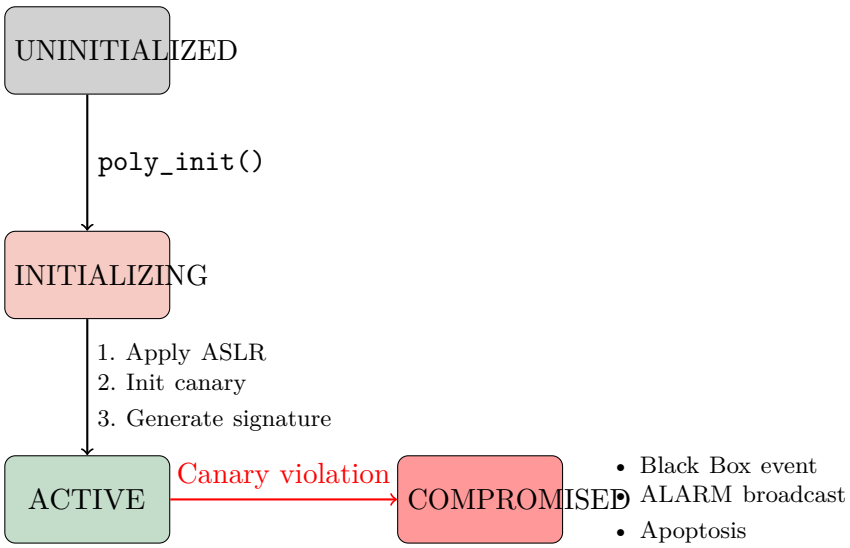
```
#define POLY_JITTER_MIN_US      10      /* Min jitter */
#define POLY_JITTER_MAX_US      100     /* Max jitter */

#define POLY_TIMING_SAFE(code) do { \
    poly_apply_jitter(); \
    code; \
    poly_apply_jitter(); \
} while(0)

/* Uso: proteger comparaciones criptograficas */
POLY_TIMING_SAFE(
    result = crypto_verify(expected, received);
);
```

Listing 9.4: Timing Jitter para Side-Channel Resistance

9.3. Flujo de Inicialización



9.4. API Principal

```
/* Inicializacion */
int poly_init(void);
```

```
3
4  /* ASLR */
5  struct poly_memory_layout* poly_apply_aslr(uint32_t entropy);
6  struct poly_memory_layout* poly_get_layout(void);
7
8  /* Firma binaria */
9  void poly_generate_signature(void);
10 struct poly_signature* poly_get_signature(void);
11 bool poly_verify_signature(struct poly_signature* sig);
12
13 /* Stack canary */
14 void poly_canary_init(void);
15 uint32_t poly_canary_get(void);
16 bool poly_canary_check(void);
17 void poly_canary_refresh(void);
18 void poly_canary_violated(void);  /* Trigger security response */
19
20 /* Timing jitter */
21 void poly_apply_jitter(void);
22 uint16_t poly_get_jitter(void);
23 void poly_randomize_jitter(void);
24
25 /* Mantenimiento */
26 void poly_tick(void);  /* Llamar desde main loop */
27 bool poly_is_active(void);
28
29 /* Debug */
30 uint8_t poly_diversity_score(void);  /* 0-255, mayor = mas diverso */
31 void poly_print_status(void);
```

Listing 9.5: API del módulo Polimorfismo

9.5. Respuesta a Violación de Canary

Cuando se detecta corrupción del stack canary:

1. **Estado** → `COMPROMISED`
2. **Black Box**: Registra `EVENT_CORRUPTION`
3. **Broadcast**: Emite `PHEROMONE_ALARM` (0xCADEAD00)
4. **Apoptosis**: El nodo se auto-destruye para proteger al swarm

```
1  void poly_canary_violated(void) {
2      /* Cambiar estado */
3      poly_state.state = POLY_STATE_COMPROMISED;
4      poly_state.canary.violations++;
5
6      /* Registrar en Black Box */
7      blackbox_record_event(EVENT_CORRUPTION, 0,
8          poly_state.canary.value, poly_state.canary.backup);
9
10     /* Alertar al swarm */
11     uint32_t alarm = 0xCADEAD00;
12     nert_send_critical(0x0000, PHEROMONE_ALARM, &alarm, 4);
13
```

```
    /* Apoptosis */
    hal_trigger_apoptosis();
}
```

Listing 9.6: Manejo de violación de canary

9.6. Diversity Score

El `diversity_score` cuantifica qué tan único es un nodo:

```
uint8_t poly_diversity_score(void) {
    uint8_t score = 0;

    /* ASLR contribuye bits de entropia */
    score += __builtin_popcount(layout.stack_offset & 0xFF) * 8;
    score += __builtin_popcount(layout.heap_offset & 0x3F) * 8;

    /* Canary contribuye bits unicos */
    score += __builtin_popcount(canary.value) * 4;

    /* Signature contribuye entropia */
    for (int i = 0; i < 16; i++) {
        score += __builtin_popcount(signature.bytes[i]);
    }

    return score;  /* 0-255, promedio esperado ~128 */
}
```

Listing 9.7: Cálculo del Diversity Score

Seguridad del Swarm

Con polimorfismo activo, un atacante necesitaría desarrollar un exploit diferente para cada nodo del swarm. Para 1000 nodos con 16,384 combinaciones de ASLR cada uno, la complejidad del ataque crece exponencialmente.

Capítulo 10

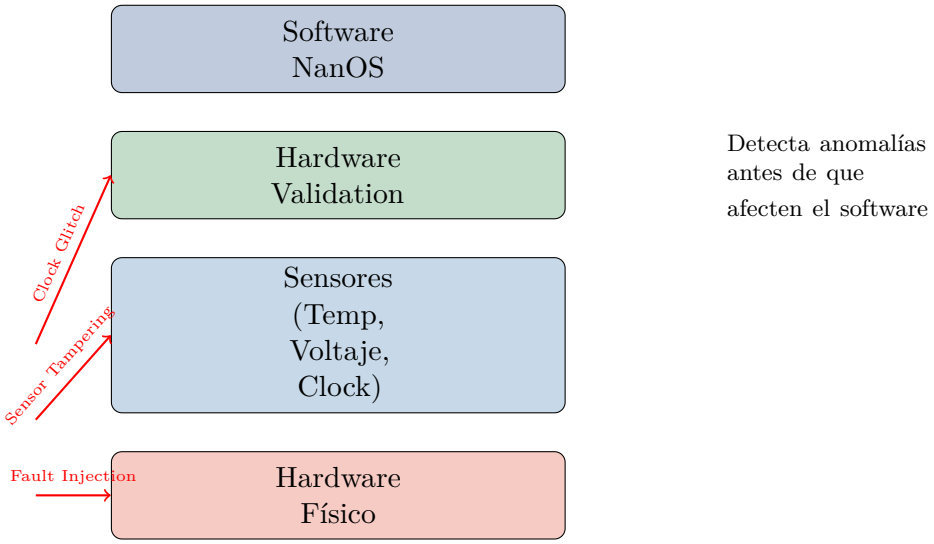
Validación de Hardware: "El Centinela del Silicio" (v0.6)

El módulo de **Validación de Hardware** ("El Centinela del Silicio") proporciona monitoreo continuo de la integridad del hardware para detectar manipulación física, sensores comprometidos y ataques de fault injection.

10.1. Motivación

El software puede ser engañado, pero el hardware torturado revela la verdad:

- **Glitching de reloj**: Manipulación de frecuencia para saltar instrucciones
- **Fault injection**: Picos de voltaje para corromper datos
- **Ataques térmicos**: Freeze/heat para afectar la memoria
- **Sensores falsificados**: Lecturas manipuladas de temperatura/voltaje
- **Firmware alterado**: Modificación no autorizada del código



10.2. Fases del Sistema

1. **CALIBRATING** (5s): Aprende baseline de sensores

2. **ACTIVE**: Monitoreo continuo normal
3. **SUSPICIOUS**: Anomalías detectadas, alerta elevada
4. **COMPROMISED**: Integridad fallida, apoptosis

10.3. Validación de Sensores

```
/* Temperatura operacional */
#define HWVAL_TEMP_MIN_C        (-40)   /* -40C minimo */
#define HWVAL_TEMP_MAX_C        85      /* 85C maximo */
#define HWVAL_TEMP_DELTA_MAX    10      /* Max cambio/segundo */

/* Voltaje de operacion */
#define HWVAL_VOLTAGE_MIN_MV    2700    /* 2.7V minimo */
#define HWVAL_VOLTAGE_MAX_MV    3600    /* 3.6V maximo */
#define HWVAL_VOLTAGE_DELTA_MAX 100     /* Max cambio/tick (mV) */

/* Tolerancia de reloj */
#define HWVAL_CLOCK_TOLERANCE_PCT   5   /* 5% drift permitido */
#define HWVAL_CLOCK_GLITCH_THRESH   3   /* Glitches antes de alarma */
```

Listing 10.1: Límites físicos de sensores

10.4. Canaries de Memoria

```
#define HWVAL_MEM_CANARY_COUNT  4
#define HWVAL_MEM_CANARY_MAGIC  0xC0DEBABE

struct hwval_mem_canary {
    uint32_t *address;      /* Ubicacion del canary */
    uint32_t expected;      /* Valor esperado */
    uint8_t  region;        /* RAM/FLASH/etc */
    uint8_t  violated;      /* Corrupcion detectada? */
};

/* Registrar ubicacion critica */
int idx = hwval_register_canary(&critical_var, HWVAL_REGION_RAM);

/* Verificacion periodica */
if (hwval_check_canaries() > 0) {
    /* Memoria corrompida! */
}
```

Listing 10.2: Protección con Memory Canaries

10.5. Integridad de Flash

```
struct hwval_flash_block {
    uint32_t start_addr;    /* Direccion inicio */
    uint32_t size;          /* Tamano del bloque */
    uint32_t crc;           /* CRC esperado */
};

```

```c
/* Registrar bloque de codigo critico */
hwval_register_flash_block(0x08000000, 0x10000);

/* Verificar integridad */
if (!hwval_verify_flash_block(0)) {
    /* Firmware alterado! */
}
```

Listing 10.3: Verificación CRC de Flash

10.6. Tipos de Violación

Código	Tipo	Severidad	Descripción
0x01	TEMP_RANGE	CRITICAL	Temperatura fuera de límites
0x04	VOLTAGE_SPIKE	CRITICAL	Glitch de voltaje detectado
0x06	CLOCK_GLITCH	CRITICAL	Glitching de reloj
0x07	MEM_CANARY	FATAL	Canary de memoria corrompido
0x08	FLASH_CRC	FATAL	CRC de flash no coincide
0x0A	SENSOR_STUCK	WARNING	Sensor no responde

Cuadro 10.1: Tipos de violación de hardware

10.7. Trust Score

Cada nodo calcula un "Trust Score" de hardware (0-255):

```c
uint8_t hwval_trust_score(void) {
    if (state == HWVAL_STATE_COMPROMISED) return 0;

    uint8_t score = 255 - anomaly_score;

    /* Penalizar por violaciones recientes */
    score -= total_violations * 10;

    return score;
}
```

Listing 10.4: Cálculo del Trust Score

10.8. API Principal

```c
/* Inicializacion */
int hwval_init(void);

/* Tick periodico (llamar cada 100ms) */
void hwval_tick(void);

/* Forzar verificacion completa */
uint8_t hwval_check_now(void);

/* Hardware confiable? */
bool hwval_is_trusted(void);
```

```
12
13 /* Registrar canary de memoria */
14 int hwval_register_canary(uint32_t *address, uint8_t region);
15
16 /* Registrar bloque flash para CRC */
17 int hwval_register_flash_block(uint32_t start, uint32_t size);
18
19 /* Obtener trust score */
20 uint8_t hwval_trust_score(void);
21
22 /* Debug */
23 void hwval_print_status(void);
```

Listing 10.5: API del módulo Hardware Validation

Defensa en Profundidad

Hardware Validation complementa las otras capas de seguridad:

- **AIS** detecta anomalías de red
- **Polimorfismo** hace únicos los binarios
- **Hardware Validation** detecta ataques físicos

Un atacante debe superar todas las capas para comprometer un nodo.

Capítulo 11

Sintonización Genética (v0.7)

11.1. Visión General

El sistema de **Sintonización Genética** permite a la Queen (micrOS) optimizar automáticamente los parámetros del protocolo NERT utilizando algoritmos genéticos. Los Workers reciben configuraciones de genoma y reportan métricas de rendimiento para que la Queen evalúe el fitness de cada configuración.

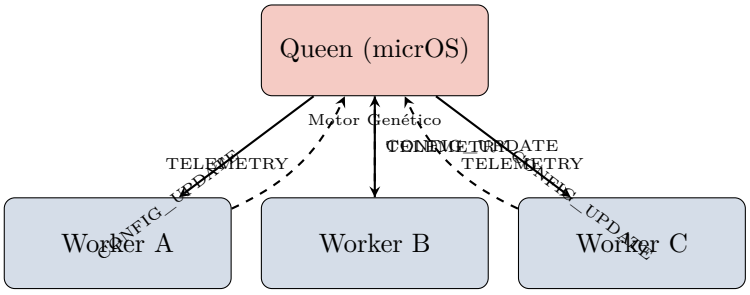


Figura 11.1: Flujo de Sintonización Genética entre Queen y Workers

11.2. Estructura del Genoma

Cada genoma contiene 32 bytes de parámetros optimizables:

```
struct nert_genome {
    uint16_t genome_id;              // ID unico
    uint8_t  generation;             // Generacion
    uint8_t  version;                // Version estructura

    // Timing Genes
    uint16_t tick_interval_ms;       // [10-200]
    uint16_t retry_timeout_ms;       // [50-1000]
    uint8_t  max_retries;            // [2-15]
    uint8_t  window_size;            // [1-8]

    // Gossip Genes
    uint8_t  gossip_prob_base;       // [50-100]
    uint8_t  gossip_prob_decay;      // [10-60]
    uint8_t  jitter_min_ms;          // [5-50]
    uint8_t  jitter_max_ms;          // [50-200]

    // Security Genes
    uint8_t  rate_bucket_capacity;   // [5-50]
```

```
    uint8_t  reputation_warn;        // [50-90]
    uint8_t  reputation_ban;         // [10-40]

    uint16_t fitness_score;          // Calculado [0-10000]
    uint16_t checksum;               // CRC16
};
```

Listing 11.1: Estructura del genoma NERT

11.3. Recepción de Configuración

Cuando un Worker recibe un pheromone `CONFIG_UPDATE` de la Queen:

1. Verificar HMAC (solo Queen puede enviar configuraciones)
2. Validar CRC16 del genoma
3. Verificar que todos los parámetros estén en rango
4. Aplicar configuración gradualmente (evitar cambios bruscos)
5. Registrar evento en BlackBox

```
void genetic_process_config_update(const struct nanos_pheromone *pkt) {
    struct nert_genome *genome = (struct nert_genome*)pkt->payload;

    // Verificar checksum
    if (!genetic_verify_checksum(genome)) {
        blackbox_record_event(EVENT_GENETIC_BAD_CHECKSUM,
            genome->genome_id);
        return;
    }

    // Aplicar configuracion
    if (genetic_apply_genome(genome) == 0) {
        g_state.current_genome_id = genome->genome_id;
        g_state.genome_generation = genome->generation;
        blackbox_record_event(EVENT_GENETIC_CONFIG_APPLIED,
            genome->genome_id);
    }
}
```

Listing 11.2: Procesamiento de CONFIG_UPDATE

11.4. Reporte de Telemetría

Periódicamente (cada 60 segundos), los Workers reportan métricas a la Queen:

```
struct genetic_telemetry {
    uint16_t genome_id;              // Genoma actual
    uint16_t packets_tx;             // Paquetes enviados
    uint16_t packets_rx;             // Paquetes recibidos
    uint16_t packets_dropped;        // Paquetes perdidos
    uint16_t avg_rtt_ms;             // RTT promedio
    uint8_t  neighbors_active;       // Vecinos activos
    uint8_t  security_violations;    // Violaciones de seguridad
```

```
    uint16_t uptime_minutes;          // Tiempo activo
};
```

Listing 11.3: Estructura de telemetría

11.5. API del Receptor Genético

```
// Inicializar receptor
void genetic_receiver_init(void);

// Procesar paquete CONFIG_UPDATE
void genetic_process_config_update(const struct nanos_pheromone *pkt);

// Enviar telemetria a Queen
void genetic_send_telemetry(void);

// Obtener genoma actual
uint16_t genetic_get_current_genome_id(void);

// Tick periodico (llamar desde main loop)
void genetic_tick(void);
```

Listing 11.4: API del receptor genético en Worker

Capítulo 12

Nodos Judas: Honeypots Activos (v0.7)

12.1. Concepto

Los **Nodos Judas** son Workers que, al detectar una intrusión, fingen vulnerabilidad para capturar los payloads de los atacantes antes de autodestruirse. Este sistema transforma nodos comprometidos en trampas que generan inteligencia de amenazas.

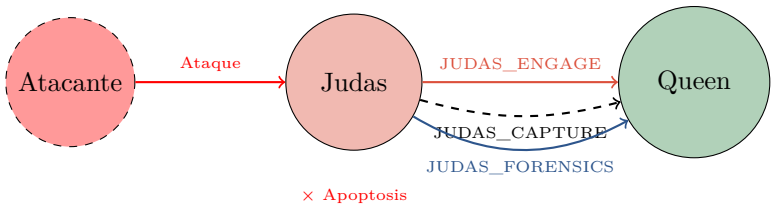


Figura 12.1: Flujo de operación de un Nodo Judas

12.2. Máquina de Estados

```
typedef enum {
    JUDAS_STATE_DORMANT,          // Comportamiento normal
    JUDAS_STATE_SUSPICIOUS,       // Intrusion detectada, evaluando
    JUDAS_STATE_ENGAGING,         // Fingiendo vulnerabilidad
    JUDAS_STATE_CAPTURING,        // Grabando payload del atacante
    JUDAS_STATE_DETONATING        // Apoptosis con forensics
} judas_state_t;
```

Listing 12.1: Estados del Nodo Judas

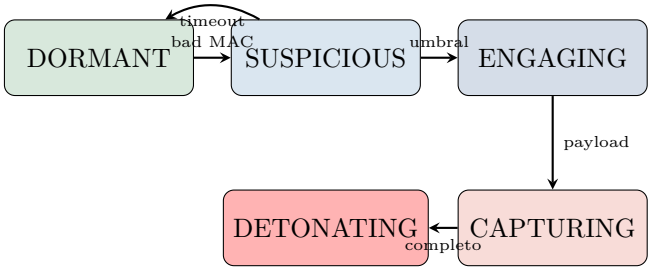


Figura 12.2: Máquina de estados del Nodo Judas

12.3. Triggers de Activación

```c
struct judas_trigger_config {
    uint8_t  bad_mac_threshold;      // MACs fallidos para activar [3-10]
    uint8_t  replay_threshold;       // Replays para activar [2-5]
    uint8_t  probe_threshold;        // Probes anomalos [5-15]
    uint16_t engage_timeout_ms;      // Timeout en ENGAGING [5000-30000]
    uint16_t capture_max_bytes;      // Max bytes a capturar [256-1024]
    uint8_t  auto_detonate;          // Detonar automaticamente
};
```

Listing 12.2: Configuración de triggers Judas

12.4. Estructura de Captura

```c
struct judas_capture {
    uint32_t attacker_id;           // Node ID del atacante
    uint8_t  attacker_mac[6];       // MAC del atacante
    uint32_t engage_tick;           // Cuando se engancho
    uint32_t capture_tick;          // Cuando se capturo

    // Tecnicas observadas
    uint8_t  bad_mac_count;         // MACs invalidos
    uint8_t  replay_count;          // Intentos de replay
    uint8_t  probe_count;           // Probes anomalos
    uint8_t  injection_attempts;    // Intentos de inyeccion

    // Payload capturado
    uint16_t payload_len;
    uint8_t  payload[512];          // Maximo payload del atacante

    // Hash para deduplicacion
    uint32_t payload_hash;
};
```

Listing 12.3: Datos capturados por Judas

12.5. Integración con AIS

El sistema Judas se integra con el Sistema Inmune Artificial:

```c
// En ais_on_anomaly():
if (g_state.judas_mode_enabled) {
    if (detect_type == AIS_DETECT_INJECTION ||
        detect_type == AIS_DETECT_PROBE) {
        // Transicionar a modo Judas en lugar de rechazar
        judas_enter_suspicious(source_node, detect_type);
        return;  // No alertar al atacante
    }
}
```

Listing 12.4: Integración Judas-AIS

12.6. Transmisión de Forensics

Antes de la apoptosis, el nodo Judas transmite toda la evidencia:

```c
void judas_detonate(void) {
    // 1. Notificar a Queen
    struct judas_capture *cap = &g_judas.capture;
    pheromone_send_judas_capture(cap);

    // 2. Transmitir BlackBox completo
    blackbox_emit_last_will();

    // 3. Transmitir forensics especifico
    pheromone_send_judas_forensics(&g_judas);

    // 4. Apoptosis
    nanos_die(DEATH_REASON_JUDAS_DETONATE);
}
```

Listing 12.5: Transmisión de forensics antes de morir

12.7. API del Sistema Judas

```c
// Inicializar sistema Judas
void judas_init(void);

// Habilitar/deshabilitar modo Judas
void judas_set_enabled(bool enabled);

// Transicion a estado sospechoso
void judas_enter_suspicious(uint32_t source, uint8_t trigger_type);

// Registrar actividad del atacante
void judas_record_activity(const void *data, uint16_t len);

// Forzar detonacion
void judas_detonate(void);

// Tick periodico
void judas_tick(void);

// Estado actual
judas_state_t judas_get_state(void);
```

Listing 12.6: API del sistema Judas

Capítulo 13

Canales Encubiertos (v0.7)

13.1. Visión General

Los **Canales Encubiertos** permiten comunicación física entre nodos utilizando side-channels como LEDs y buzzers. Esto habilita la comunicación en entornos air-gapped donde la red tradicional no está disponible.

Figura 13.1: Comunicación por canal encubierto

13.2. Canal Óptico (LED-to-Light Sensor)

13.2.1. Características

- **Transmisor**: LED controlado por PWM
- **Receptor**: Fotorresistor o fotodiodo conectado a ADC
- **Modulación**: On-Off Keying (OOK) o Manchester
- **Velocidad**: 10–100 bps
- **Alcance**: 1–5 metros (dependiente de luz ambiental)

13.2.2. Codificación Manchester

Manchester encoding proporciona auto-sincronización:

```
// Bit 0: transicion HIGH->LOW en mitad del periodo
// Bit 1: transicion LOW->HIGH en mitad del periodo

void covert_manchester_encode_bit(uint8_t bit) {
    if (bit) {
        // Bit 1: LOW primera mitad, HIGH segunda mitad
        led_set(0);
        delay_us(BIT_PERIOD_US / 2);
        led_set(1);
        delay_us(BIT_PERIOD_US / 2);
```

```
    } else {
        // Bit 0: HIGH primera mitad, LOW segunda mitad
        led_set(1);
        delay_us(BIT_PERIOD_US / 2);
        led_set(0);
        delay_us(BIT_PERIOD_US / 2);
    }
}
```

Listing 13.1: Codificación Manchester

13.3. Canal Acústico (Buzzer-to-Mic)

13.3.1. Características

- **Transmisor**: Buzzer piezoeléctrico con PWM
- **Receptor**: Micrófono analógico o I2S
- **Modulación**: FSK (Frequency Shift Keying)
- **Frecuencias**: 1kHz/2kHz (audible) o 18kHz/20kHz (ultrasónico)
- **Velocidad**: 50–200 bps

13.3.2. Modulación FSK

```
#define FREQ_ZERO   1000    // Hz para bit 0
#define FREQ_ONE    2000    // Hz para bit 1
#define BIT_TIME_MS 20      // Duracion de cada bit

void covert_fsk_send_bit(uint8_t bit) {
    uint16_t freq = bit ? FREQ_ONE : FREQ_ZERO;
    buzzer_set_frequency(freq);
    buzzer_enable();
    delay_ms(BIT_TIME_MS);
    buzzer_disable();
}
```

Listing 13.2: Modulación FSK para canal acústico

13.4. Estructura del Frame

```
#define COVERT_SYNC_PATTERN     0xAA
#define COVERT_CHANNEL_OPTICAL  0x01
#define COVERT_CHANNEL_ACOUSTIC 0x02

struct covert_frame {
    uint8_t  sync_pattern;      // 0xAA para sincronizacion
    uint8_t  channel_type;      // OPTICAL o ACOUSTIC
    uint8_t  payload_len;       // 1-16 bytes
    uint8_t  payload[16];       // Datos
    uint8_t  crc8;              // Checksum
};
```

Listing 13.3: Estructura del frame covert

13.5. API de Canales Encubiertos

```
// Inicializar transmisor
void covert_tx_init(uint8_t channel);

// Enviar datos
int covert_tx_send(const void *data, uint8_t len);

// Inicializar receptor
void covert_rx_init(uint8_t channel, uint8_t gpio_sensor);

// Recibir datos (bloqueante con timeout)
int covert_rx_receive(void *buffer, uint8_t max_len, uint32_t
    timeout_ms);

// Calibrar receptor
void covert_rx_calibrate(void);

// Obtener estadisticas
void covert_get_stats(struct covert_stats *stats);
```

Listing 13.4: API de canales encubiertos

13.6. Consideraciones de Seguridad

Seguridad de Canales Encubiertos

- Los datos transmitidos por canal encubierto **deben** encriptarse con la misma clave de sesión NERT
- Incluir sequence number para prevenir replay attacks
- El payload máximo es 16 bytes por frame
- Los canales encubiertos son detectables; usar con discreción

```
int covert_send_secure(const void *data, uint8_t len) {
    uint8_t encrypted[16];

    // Encriptar con ChaCha8
    chacha8_encrypt(g_state.session_key, g_state.covert_nonce,
                    data, encrypted, len);

    // Incrementar nonce
    g_state.covert_nonce++;

    // Enviar por canal encubierto
    return covert_tx_send(encrypted, len);
}
```

Listing 13.5: Encriptación de datos covert

Parte II

Protocolo NERT

Capítulo 14

Visión General de NERT

14.1. Motivación

Los protocolos tradicionales no satisfacen las necesidades de nodos desechables:

Característica	**TCP**	**UDP**	**NERT**
Confiabilidad	Siempre	Nunca	**Selectiva**
Encriptación	Opcional	No	**Obligatoria**
Handshake	3-way	Ninguno	**2-way**
FEC	No	No	**Opcional**
Multi-path	No	No	**Sí**
RAM/conexión	∼2KB	∼0	**∼100B**
Optimizado para efímeros	No	Parcial	**Sí**

Cuadro 14.1: Comparación de NERT con protocolos tradicionales

14.2. Clases de Confiabilidad

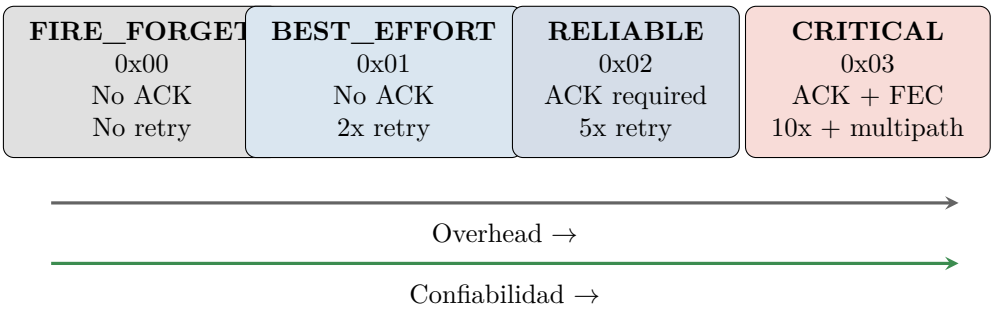


Figura 14.1: Clases de confiabilidad de NERT

Capítulo 15

Formato de Paquete NERT

15.1. Header Estándar (20 bytes)

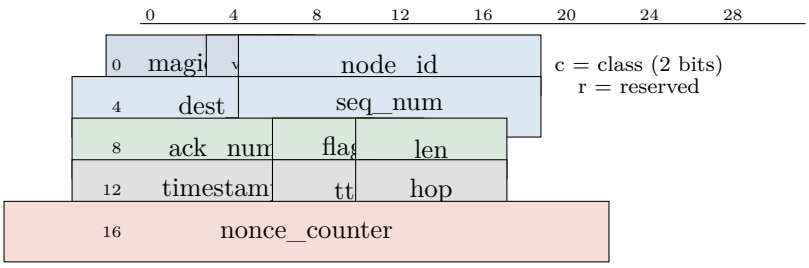


Figura 15.1: Header NERT estándar de 20 bytes

15.2. Campo de Flags

Bit	Nombre	Descripción
0	SYN	Inicio de conexión
1	ACK	Acknowledgment válido
2	FIN	Fin de conexión
3	RST	Reset/abort
4	ENC	Payload encriptado (siempre 1)
5	FEC	Incluye bloque FEC
6	FRAG	Paquete fragmentado
7	MPATH	Multi-path habilitado

Cuadro 15.1: Definición de bits del campo flags

15.3. Estructura Completa del Paquete

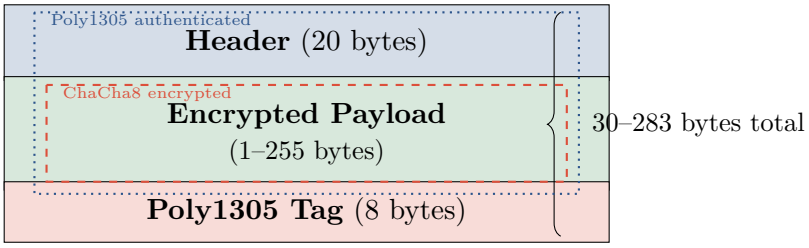


Figura 15.2: Estructura completa del paquete NERT

Capítulo 16

Criptografía

16.1. Algoritmos Utilizados

Función	Algoritmo	Parámetros
Encriptación	ChaCha8	256-bit key, 96-bit nonce
Autenticación	Poly1305	256-bit key, truncated to 64-bit
Key Derivation	ChaCha8-based PRF	Master key + epoch

Cuadro 16.1: Algoritmos criptográficos de NERT

16.2. Derivación de Claves

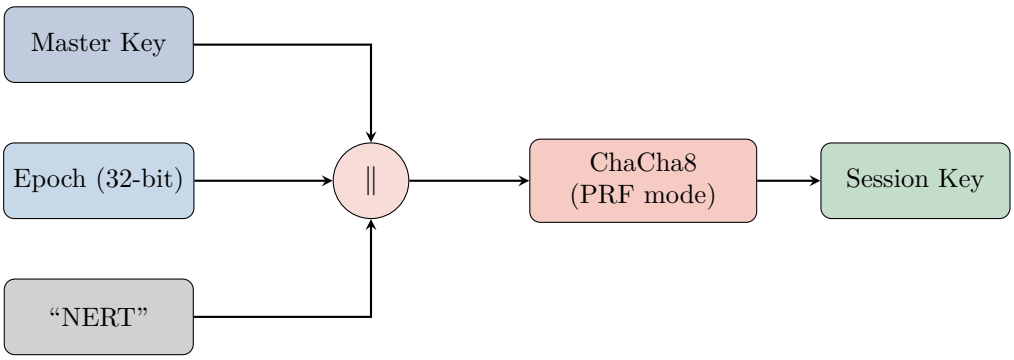
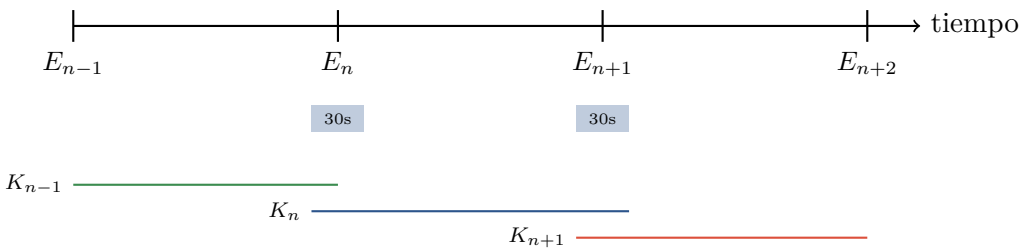


Figura 16.1: Proceso de derivación de clave de sesión

16.3. Ventana de Gracia para Rotación de Claves



Ventana de gracia: acepta claves de épocas adyacentes

Figura 16.2: Mecanismo de ventana de gracia para rotación de claves

Problema de Sincronización

Sin sincronización de tiempo, nodos con relojes desincronizados podrían rechazar paquetes válidos en el cambio de época. La ventana de gracia de 30 segundos permite tolerancia a drift de hasta ±30s entre nodos.

16.4. Construcción del Nonce

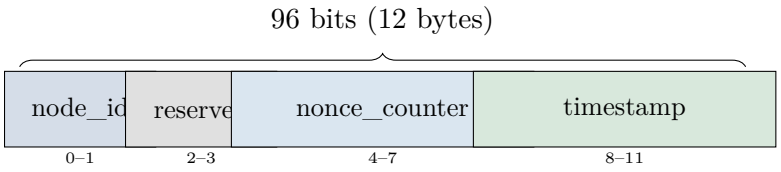


Figura 16.3: Estructura del nonce de 96 bits

Capítulo 17

Mecanismos de Confiabilidad

17.1. Máquina de Estados de Conexión

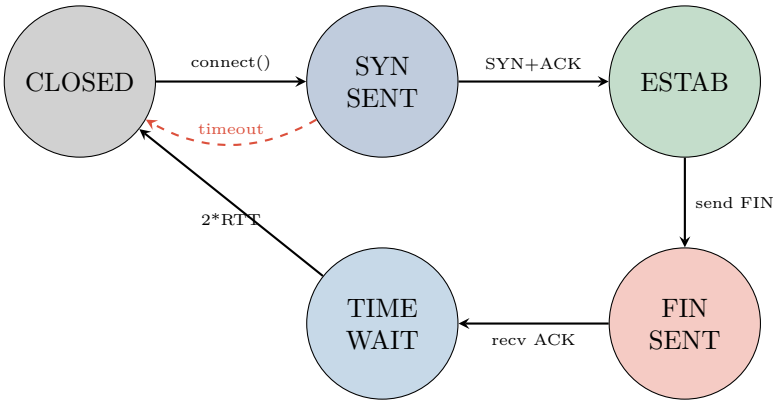


Figura 17.1: Máquina de estados de conexión NERT (simplificada)

17.2. Two-Way Handshake

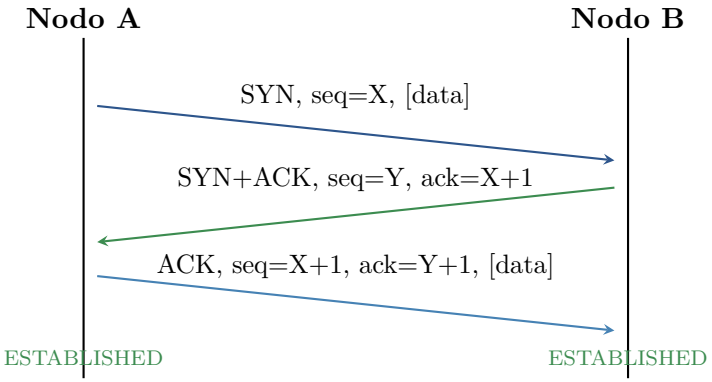


Figura 17.2: Handshake de dos vías de NERT

17.3. Selective ACK (SACK)

El SACK permite indicar eficientemente qué paquetes se han recibido:

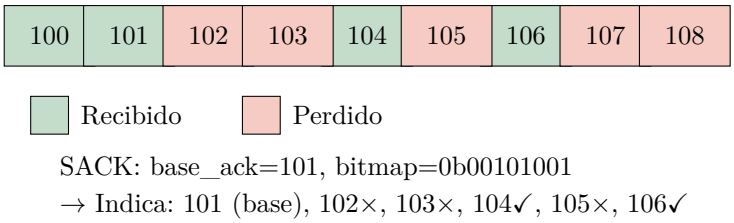


Figura 17.3: Ejemplo de Selective ACK

17.4. Retransmisión con Backoff Exponencial

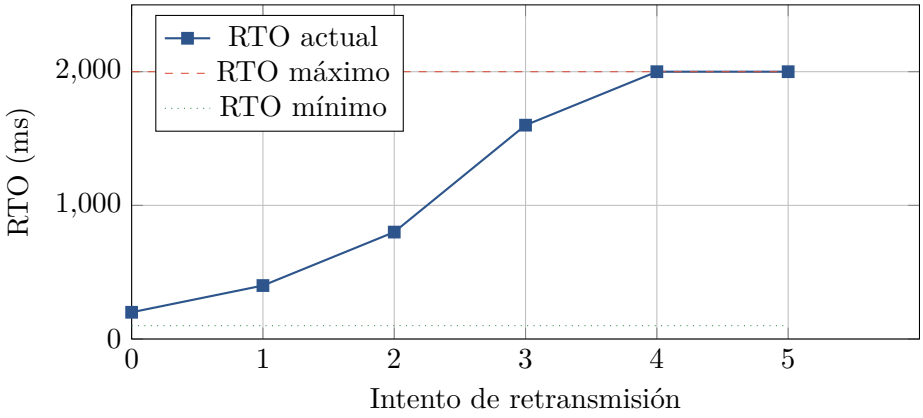


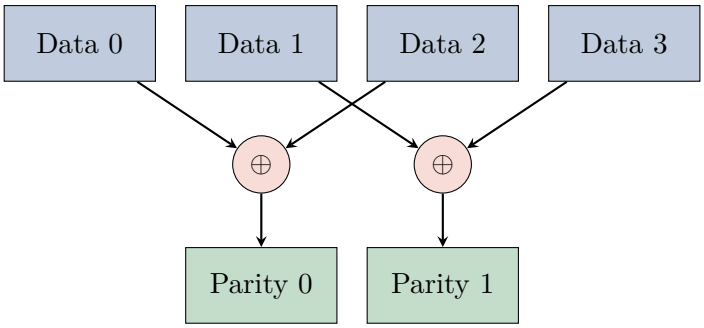
Figura 17.4: Crecimiento del RTO con backoff exponencial

Capítulo 18

Forward Error Correction

18.1. Esquema XOR Parity

Para la clase CRITICAL, NERT utiliza FEC basado en XOR:



$P_0 = D_0 \oplus D_2$

$P_1 = D_1 \oplus D_3$

Recuperación: $D_0 = P_0 \oplus D_2$, $D_2 = P_0 \oplus D_0$, etc.

Figura 18.1: Esquema FEC con paridad XOR

18.2. Capacidad de Recuperación

Pérdida	**Recuperable?**
1 data shard cualquiera	✓ Sí
2 data shards (grupos diferentes)	✓ Sí
2 data shards (mismo grupo)	× No
3+ data shards	× No

Cuadro 18.1: Capacidad de recuperación del FEC

Capítulo 19

Multi-Path Transmission

19.1. Selección de Rutas

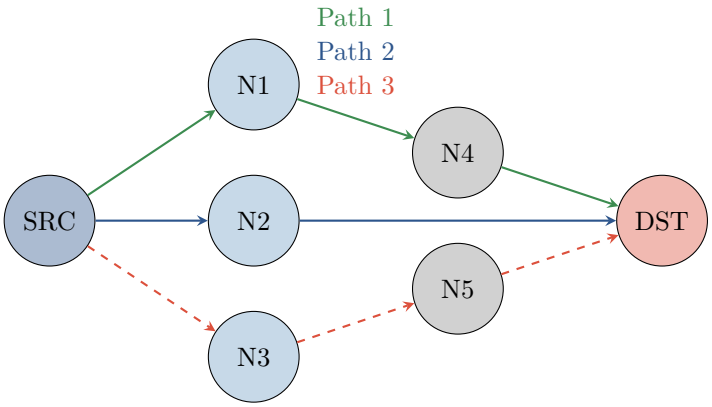


Figura 19.1: Transmisión multi-path con rutas diversas

Capítulo 20

Hive Bridge: Integración con micrOS

20.1. Arquitectura del Enjambre Híbrido

El **Hive Bridge** permite la comunicación bidireccional entre nodos NanOS (obreras) y micrOS (reina). Esta arquitectura híbrida combina:

- **micrOS**: Sistema operativo distribuido completo actuando como reina
- **NanOS**: Nodos desechables actuando como obreras del enjambre

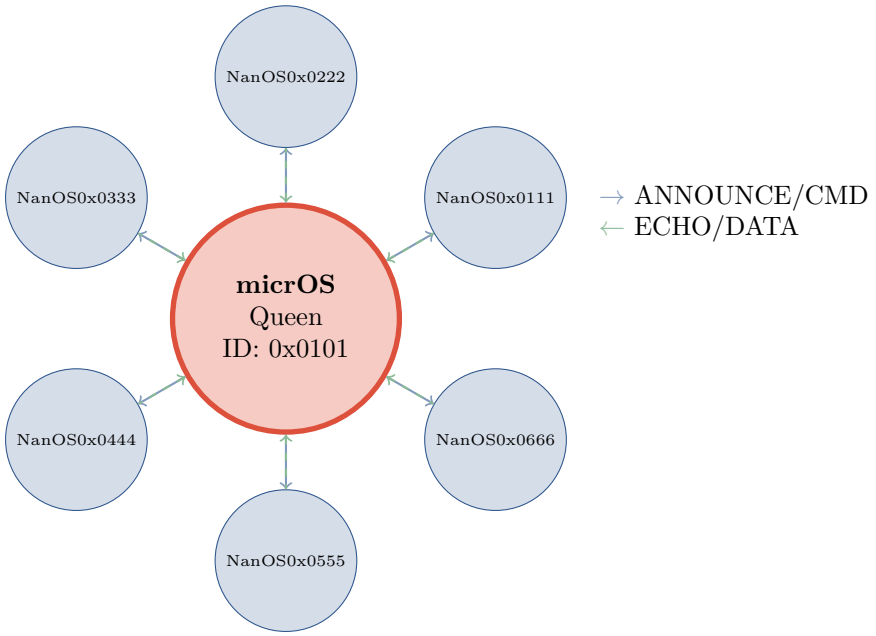


Figura 20.1: Arquitectura Hive: micrOS como Queen coordinando nodos NanOS

20.2. Detección de la Reina

Cuando un nodo NanOS detecta un paquete ANNOUNCE de micrOS:

```
void on_announce_received(uint16_t sender_id, const void *data, uint8_t
    len) {
    announce_data_t *announce = (announce_data_t*)data;

    // micrOS se anuncia con role = ROLE_QUEEN y distance = 0
```

```
    if (announce->role == ROLE_QUEEN && announce->distance == 0) {
        // Encontramos la reina!
        g_queen_id = sender_id;
        g_distance_to_queen = 1;  // Directo a la reina

        LOG_INFO("HIVE", "Queen detected: 0x%04x", sender_id);

        // Actualizar gradiente
        update_gradient(sender_id, 0);
    }
}
```

Listing 20.1: Detección de micrOS como reina

20.3. Servicio NERT en micrOS

micrOS implementa un servicio NERT que:

1. **PHY Adapter**: Traduce entre el stack DOD de micrOS y NERT
2. **Swarm State**: Mantiene estado de todos los nodos del enjambre
3. **Polling**: Procesa pheromones incluso sin scheduler activo

```
typedef struct {
    swarm_node_entry_t nodes[32];  /* Nodos conocidos */
    uint32_t node_count;
    swarm_metrics_t metrics;       /* Metricas agregadas */
    uint16_t local_node_id;        /* ID de micrOS (queen) */
    uint8_t  initialized;
    uint8_t  running;
} swarm_state_t;
```

Listing 20.2: Estructura del estado del swarm en micrOS

20.4. Comunicación Bidireccional

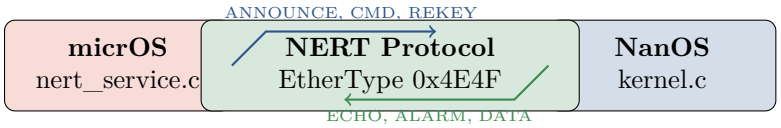


Figura 20.2: Flujo de comunicación entre micrOS y NanOS vía NERT

20.5. Polling en micrOS

Cuando el scheduler de micrOS no está activo, el shell debe pollear manualmente:

```
// En micrOS shell.c
void shell_run(void) {
    while (g_running) {
        char c = keyboard_getchar_nonblock();
        if (c != 0) {
            shell_input_char(c);
```

```
        }

        // Critico: Poll NERT para comunicacion con swarm
        nert_service_poll();

        shell_cpu_halt();
    }
}
```

Listing 20.3: Polling de NERT en el shell de micrOS

Importante

Sin el polling manual, micrOS no procesará pheromones entrantes de los nodos NanOS cuando el scheduler no esté realizando context switches.

20.6. Verificación de Conectividad

Para verificar que NanOS detecta a micrOS como reina:

```
[NERT] RX pheromone: type=0x01 from=0x0101 dist=0
[HIVE] Queen detected: 0x0101
[GRADIENT] Updated: queen=0x0101 distance=1

# Status command shows:
> status
  Node ID:   0x0A01
  Role:      WORKER
  Queen:     0x0101
  Distance:  1
  Uptime:    42s
```

Listing 20.4: Salida esperada en NanOS

20.7. Comandos de la Reina

micrOS puede enviar comandos a los nodos NanOS mediante pheromones COMMAND:

Código	Comando	Descripción
0x01	ROLE_ASSIGN	Cambiar rol del nodo
0x02	TASK_EXEC	Ejecutar tarea específica
0x03	CONFIG_SET	Modificar configuración
0x10	STATUS_REQ	Solicitar status del nodo
0x13	DIE	Ordenar apoptosis

Cuadro 20.1: Comandos de la reina a nodos worker

Parte III

Guía de Implementación

Capítulo 21

Compilación y Despliegue

21.1. Requisitos

Componente	Versión Mínima
GCC (x86)	9.0
ARM GCC	arm-none-eabi-gcc 10.0
QEMU	6.0
PlatformIO	6.0 (para ESP32)
Make	4.0

Cuadro 21.1: Requisitos de compilación

21.2. Compilación x86

```
# Compilar kernel
make clean
make PLATFORM=x86

# Ejecutar en QEMU (3 nodos)
./scripts/run_swarm.sh 3
```

Listing 21.1: Compilación para QEMU x86

21.3. Compilación ARM

```
# Compilar para Stellaris
make PLATFORM=arm

# Ejecutar en QEMU
qemu-system-arm -M lm3s6965evb \
    -kernel build/nanos_arm.elf \
    -nographic
```

Listing 21.2: Compilación para ARM Cortex-M3

21.4. Compilación ESP32

```
cd platforms/esp32
pio run -e esp32dev

# Flash al dispositivo
pio run -t upload
```

Listing 21.3: Compilación con PlatformIO

Capítulo 22

API de Programación

22.1. Inicialización

```c
#include "nert.h"
#include "nanos.h"

void handle_pheromone(uint16_t sender, uint8_t type,
                      const void *data, uint8_t len) {
    switch (type) {
        case PHEROMONE_HELLO:
            neighbor_update(sender, data);
            break;
        case PHEROMONE_QUEEN_CMD:
            execute_command(data);
            break;
        // ... otros tipos
    }
}

void kernel_main(void) {
    hal_init();
    nert_init();
    nert_set_receive_callback(handle_pheromone);

    while (1) {
        nert_process_incoming();
        nert_timer_tick();
        nert_check_key_rotation();

        // Logica de aplicacion
        process_role_logic();

        hal_cpu_idle();
    }
}
```

Listing 22.1: Inicialización típica de NERT

22.2. Envío de Mensajes

```c
// Telemetria frecuente (no garantizada)
struct sensor_data sensor = {.temp = 25, .humidity = 60};
```

```c
nert_send_unreliable(0, PHEROMONE_SENSOR, &sensor, sizeof(sensor));

// Datos importantes (con retry)
struct kv_pair kv = {.key = "status", .value = "active"};
nert_send_best_effort(dest_id, PHEROMONE_KV_SET, &kv, sizeof(kv));

// Comando que requiere ACK
struct task_cmd task = {.action = ACTION_EXPLORE, .area = 5};
int conn = nert_send_reliable(worker_id, PHEROMONE_TASK,
                              &task, sizeof(task));

// Comando critico (FEC + multipath)
struct die_cmd die = {.target = rogue_node, .reason = SECURITY};
nert_send_critical(target_id, PHEROMONE_DIE, &die, sizeof(die));
```

Listing 22.2: Ejemplos de envío con diferentes clases

Capítulo 23

Debugging y Monitoreo

23.1. Estadísticas de NERT

```c
const struct nert_stats *stats = nert_get_stats();

printf("TX: %lu packets, %lu bytes\n",
       stats->tx_packets, stats->tx_bytes);
printf("RX: %lu packets, %lu duplicates\n",
       stats->rx_packets, stats->rx_duplicates);
printf("Retransmits: %lu\n", stats->tx_retransmits);
printf("Bad MAC: %lu, Replay blocked: %lu\n",
       stats->rx_bad_mac, stats->rx_replay_blocked);
printf("RTT: avg=%u, min=%u, max=%u\n",
       stats->avg_rtt, stats->min_rtt, stats->max_rtt);
```

Listing 23.1: Acceso a estadísticas de protocolo

23.2. Métricas Clave

Métrica	Valor Normal	Alerta Si
Duplicates/RX	$< 5\,\%$	$> 20\,\%$
Bad MAC rate	$< 0.1\,\%$	$> 1\,\%$
Retransmit rate	$< 10\,\%$	$> 30\,\%$
Avg RTT	$<$ 100ms	$>$ 500ms
Connection timeouts	0	$>$ 5/min

Cuadro 23.1: Umbrales de métricas para monitoreo

Apéndice A

Constantes de Configuración

Constante	Valor	Descripción
NERT_MAGIC	0x4E	Byte mágico ('N')
NERT_VERSION	0x10	Versión del protocolo
NERT_KEY_SIZE	32	Tamaño de clave en bytes
NERT_NONCE_SIZE	12	Tamaño de nonce en bytes
NERT_MAC_SIZE	8	Tamaño de tag MAC
NERT_MAX_CONNECTIONS	4–8	Conexiones simultáneas
NERT_WINDOW_SIZE	2–4	Tamaño de ventana TX
NERT_RETRY_TIMEOUT_MS	200	Timeout inicial
NERT_MAX_RETRIES	5	Reintentos (RELIABLE)
NERT_MAX_RETRIES_CRITICAL	10	Reintentos (CRITICAL)
NERT_KEY_ROTATION_SEC	3600	Rotación de clave (1h)
NERT_KEY_GRACE_WINDOW_MS	30000	Ventana de gracia (30s)
BLOOM_BITS	256	Bits del Bloom filter
BLOOM_SLOTS	4	Slots rotativos
BLOOM_WINDOW_MS	500	Ventana por slot
MAX_CELL_LIFETIME	3600000	Vida máxima del nodo (ms)
HEAP_CRITICAL_PCT	90	Umbral crítico de heap
Enrutamiento Hebbiano (v0.5)		
SYNAPSE_WEIGHT_MIN	1	Peso sináptico mínimo
SYNAPSE_WEIGHT_MAX	255	Peso sináptico máximo
SYNAPSE_WEIGHT_INITIAL	128	Peso inicial (neutral)
SYNAPSE_WEIGHT_THRESHOLD	32	Umbral para ruta saludable
SYNAPSE_LTP_INCREMENT	15	Recompensa por éxito (LTP)
SYNAPSE_LTD_DECREMENT	40	Castigo por fallo (LTD)
SYNAPSE_STDP_BONUS	5	Bonus por respuesta rápida
SYNAPSE_STDP_WINDOW_MS	100	Ventana STDP (ms)
SYNAPSE_DECAY_INTERVAL_MS	5000	Intervalo de decaimiento
SYNAPSE_DECAY_AMOUNT	1	Decremento por intervalo
Stigmergia (v0.5)		
STIGMERGIA_DANGER	0	Tipo: zona de peligro
STIGMERGIA_QUEEN	1	Tipo: camino a reina
STIGMERGIA_RESOURCE	2	Tipo: marcador de recurso
STIGMERGIA_AVOID	3	Tipo: zona a evitar

Continúa en la siguiente página...

Constante	Valor	Descripción
STIGMERGIA_INTENSITY_MAX	15	Intensidad máxima (4 bits)
STIGMERGIA_DECAY_INTERVAL_MS	1000	Decaimiento cada 1s
STIGMERGIA_DECAY_AMOUNT	1	Resta 1 por intervalo
STIGMERGIA_PROPAGATE_THRESHOLD	6	Min intensidad para propagar
STIGMERGIA_PROPAGATE_DECAY	3	Decremento al propagar
STIGMERGIA_DANGER_COST_MULT	8	Multiplicador costo DANGER
STIGMERGIA_AVOID_COST_MULT	4	Multiplicador costo AVOID
STIGMERGIA_QUEEN_COST_BONUS	2	Bonus atracción QUEEN
STIGMERGIA_SIZE	16	Grid 16x16 (escala 2:1)
Black Box (v0.5)		
BLACKBOX_MAX_WILLS	8	Testamentos almacenados
DEATH_NATURAL	0x00	Muerte natural (timeout)
DEATH_HEAP_EXHAUSTED	0x01	Sin memoria
DEATH_CORRUPTION	0x02	Corrupción detectada
DEATH_ATTACK_DETECTED	0x03	Bajo ataque
DEATH_QUEEN_ORDER	0x04	Orden de la reina
DEATH_ISOLATION	0x05	Sin contacto
EVENT_BAD_MAC	0x01	MAC inválido
EVENT_REPLAY	0x02	Intento de replay
EVENT_RATE_LIMIT	0x03	Rate limit excedido
EVENT_BLACKLIST	0x04	Nodo en blacklist
EVENT_JAMMING	0x05	Jamming detectado
EVENT_CORRUPTION	0x06	Corrupción memoria
Sistema Inmune Artificial AIS (v0.6)		
AIS_DETECTOR_COUNT	16	Detectores activos
AIS_DETECTOR_SIZE	8	Bytes por patrón
AIS_AFFINITY_THRESHOLD	6	Bits contiguos para match
AIS_MATURATION_SAMPLES	32	Muestras para maduración
AIS_THYMUS_DURATION_MS	5000	Fase thymus (5s)
AIS_DETECTOR_LIFESPAN_MS	300000	Vida del detector (5min)
AIS_ANOMALY_THRESHOLD	3	Detectores para alarma
AIS_MEMORY_CELLS	4	Detectores de memoria
AIS_MEMORY_BOOST	2	Boost para memoria
AIS_DETECT_FLOOD	0x01	Detección: DoS/flooding
AIS_DETECT_PROBE	0x02	Detección: escaneo
AIS_DETECT_REPLAY	0x03	Detección: replay
AIS_DETECT_INJECTION	0x04	Detección: inyección
AIS_DETECT_BEHAVIORAL	0x05	Detección: comportamiento
AIS_DETECT_SYBIL	0x06	Detección: sybil
AIS_DETECT_UNKNOWN	0xFF	Detección: 0-day
EVENT_AIS_DETECTOR_MATCH	0x10	Detector matcheó
EVENT_AIS_THYMUS_COMPLETE	0x11	Maduración completa
EVENT_AIS_MEMORY_PROMOTE	0x12	Promovido a memoria
EVENT_AIS_ANOMALY_ALERT	0x13	Alerta de anomalía
Polimorfismo de Código "El Camaleón" (v0.6)		
POLY_ASLR_STACK_BITS	8	Bits de entropía stack (256 bases)
POLY_ASLR_HEAP_BITS	6	Bits de entropía heap (64 bases)

Continúa en la siguiente página...

Constante	Valor	Descripción
POLY_ASLR_ALIGNMENT	16	Alineación ARM (bytes)
POLY_STACK_OFFSET_MAX	4096	Max offset stack (bytes)
POLY_HEAP_OFFSET_MAX	1024	Max offset heap (bytes)
POLY_CANARY_SIZE	4	Tamaño del canary (32 bits)
POLY_CANARY_CHECK_INTERVAL	100	Verificar cada N ticks
POLY_CANARY_REFRESH_MS	60000	Refrescar cada 60s
POLY_SIGNATURE_SIZE	16	Firma única (128 bits)
POLY_SIGNATURE_REFRESH_MS	300000	Refrescar firma cada 5min
POLY_JITTER_MIN_US	10	Jitter mínimo (μs)
POLY_JITTER_MAX_US	100	Jitter máximo (μs)
POLY_NOP_REGIONS	4	Regiones NOP randomizables
POLY_NOP_MAX_SIZE	8	NOPs máximos por región
POLY_STATE_UNINITIALIZED	0x00	Estado: sin inicializar
POLY_STATE_INITIALIZING	0x01	Estado: inicializando
POLY_STATE_ACTIVE	0x02	Estado: activo
POLY_STATE_COMPROMISED	0xFF	Estado: canary violado
Validación de Hardware "El Centinela del Silicio" (v0.6)		
HWVAL_TICK_INTERVAL_MS	100	Intervalo de verificación (ms)
HWVAL_TEMP_MIN_C	-40	Temperatura mínima (°C)
HWVAL_TEMP_MAX_C	85	Temperatura máxima (°C)
HWVAL_TEMP_DELTA_MAX	10	Max cambio temp/segundo
HWVAL_VOLTAGE_MIN_MV	2700	Voltaje mínimo (mV)
HWVAL_VOLTAGE_MAX_MV	3600	Voltaje máximo (mV)
HWVAL_VOLTAGE_DELTA_MAX	100	Max cambio voltaje/tick
HWVAL_CLOCK_TOLERANCE_PCT	5	Tolerancia drift (%)
HWVAL_CLOCK_GLITCH_THRESH	3	Glitches antes de alarma
HWVAL_MEM_CANARY_COUNT	4	Canaries de memoria
HWVAL_MEM_CANARY_MAGIC	0xC0DEBABE	Valor mágico canary
HWVAL_FLASH_CRC_BLOCKS	8	Bloques flash verificados
HWVAL_ANOMALY_THRESHOLD	3	Anomalías para crítico
HWVAL_STATE_CALIBRATING	0x01	Estado: calibrando
HWVAL_STATE_ACTIVE	0x02	Estado: activo
HWVAL_STATE_SUSPICIOUS	0x03	Estado: sospechoso
HWVAL_STATE_COMPROMISED	0xFF	Estado: comprometido
EVENT_HWVAL_TEMP_VIOLATION	0x20	Violación temperatura
EVENT_HWVAL_VOLTAGE_GLITCH	0x21	Glitch de voltaje
EVENT_HWVAL_CLOCK_ANOMALY	0x22	Anomalía de reloj
EVENT_HWVAL_MEM_CORRUPT	0x23	Memoria corrompida
EVENT_HWVAL_FLASH_TAMPER	0x24	Flash alterado

Cuadro A.1: Constantes de configuración de NanOS/NERT

Apéndice B

Glosario

AIS (v0.6)
Artificial Immune System. Sistema Inmune Artificial basado en el algoritmo de Selección Negativa. Detecta anomalías sin necesidad de firmas de ataques conocidos. "El sistema inmune no necesita saber cómo se ven los ataques — solo qué aspecto tiene 'saludable'."

Apoptosis
Muerte celular programada. En NanOS, el proceso por el cual un nodo termina su ejecución de forma controlada.

Black Box (v0.5)
Sistema de preservación de evidencia forense. Los nodos emiten un "testamento" antes de morir que sobrevive en vecinos de confianza.

Bloom Filter
Estructura de datos probabilística para pruebas de membresía con eficiencia de espacio.

Canal Encubierto (v0.7)
Covert Channel. Comunicación física a través de side-channels como LEDs (óptico) o buzzers (acústico) para entornos air-gapped donde la red tradicional no está disponible.

Época (Epoch)
Período de tiempo durante el cual una clave de sesión es válida (por defecto: 1 hora).

FEC
Forward Error Correction. Técnica para recuperar datos perdidos sin retransmisión.

Gradiente
Campo escalar que indica la distancia a la reina, usado para enrutamiento.

Genetic Tuning (v0.7)
Sintonización Genética. Sistema de auto-optimización de parámetros NERT mediante algoritmos genéticos. La Queen mantiene una población de genomas y los distribuye a sub-swarms para evaluación comparativa.

Genome (v0.7)
Estructura de 32 bytes que codifica parámetros optimizables de NERT como genes evolucionables (timing, gossip, seguridad).

Gossip Protocol
Protocolo de diseminación epidémica de información.

Grace Window
Período en límites de época donde se aceptan claves de épocas adyacentes.

Hardware Validation (v0.6)
"El Centinela del Silicio". Sistema de monitoreo continuo de integridad del hardware para detectar manipulación física, sensores comprometidos, y ataques de fault injection. Valida temperatura, voltaje, frecuencia de reloj, y memoria. "El hardware no puede mentir - pero puede ser torturado para dar falsas confesiones."

HMAC
Hash-based Message Authentication Code.

Judas Node (v0.7)
Nodo Judas. Honeypot activo que, al detectar intrusión, finge vulnerabilidad para capturar payloads de atacantes antes de autodestruirse. Transmite evidencia forense a la Queen antes de la apoptosis.

LTD
Long-Term Depression. Debilitamiento de conexiones sinápticas tras eventos negativos (timeouts, fallos de entrega).

LTP
Long-Term Potentiation. Fortalecimiento de conexiones sinápticas tras eventos positivos (ACKs exitosos).

Manchester Encoding (v0.7)
Codificación donde cada bit se representa mediante una transición en mitad del período. Proporciona auto-sincronización para canales encubiertos ópticos.

Nodo Desechable
Nodo diseñado con vida útil limitada que regenera al morir.

NERT
NanOS Ephemeral Reliable Transport.

Pheromone
Mensaje de comunicación entre nodos del enjambre.

Polimorfismo de Código (v0.6)
"El Camaleón". Sistema de diversidad binaria que hace único a cada nodo. Incluye ASLR (randomización de memoria), Stack Canaries (detección de overflow), y Binary Signatures (huella digital única). "Si un atacante compromete un nodo, ha ganado una batalla. Para comprometer el swarm, tendría que ganar miles diferentes."

SACK
Selective Acknowledgment.

STDP
Spike-Timing Dependent Plasticity. Mecanismo donde el tiempo de respuesta afecta el aprendizaje (respuestas rápidas refuerzan más).

Stigmergia (v0.5)
Coordinación indirecta a través de modificación del ambiente. En NanOS, feromonas digitales que decaen con el tiempo y modifican costos de movimiento. "Las hormigas no memorizan mapas."

Sub-Swarm (v0.7)
Grupo de Workers que comparten el mismo genoma durante un período de evaluación. Permite A/B testing evolutivo de configuraciones NERT.

Swarm
Conjunto de nodos comunicándose colectivamente.

Synaptic Weight
Peso sináptico. Valor [1-255] que representa la confiabilidad aprendida de una conexión con un vecino.

Telemetry (v0.7)
Métricas de rendimiento (throughput, latency, reliability, security, survival) enviadas periódicamente por Workers a Queen para evaluar fitness de genomas.

Sobre Este Documento

Este manual técnico fue generado como parte del proyecto NanOS v0.7. Cubre la arquitectura del sistema operativo, el protocolo NERT diseñado específicamente para comunicación entre nodos desechables, el sistema de enrutamiento Hebbiano inspirado en neurociencia, el sistema de feromonas digitales Stigmergia, la Black Box distribuida para evidencia forense, el Sistema Inmune Artificial (AIS) para detección de ataques 0-day, el sistema de Polimorfismo de Código "El Camaleón" para diversidad binaria, la Validación de Hardware "El Centinela del Silicio" para detección de manipulación física, el sistema de Sintonización Genética para auto-optimización evolutiva de parámetros NERT, los Nodos Judas como honeypots activos para captura de inteligencia de amenazas, y los Canales Encubiertos para comunicación física en entornos air-gapped.

NanOS Project
Los nodos mueren, el enjambre vive.
2026