Maté: A Tiny Virtual Machine for Sensor Networks

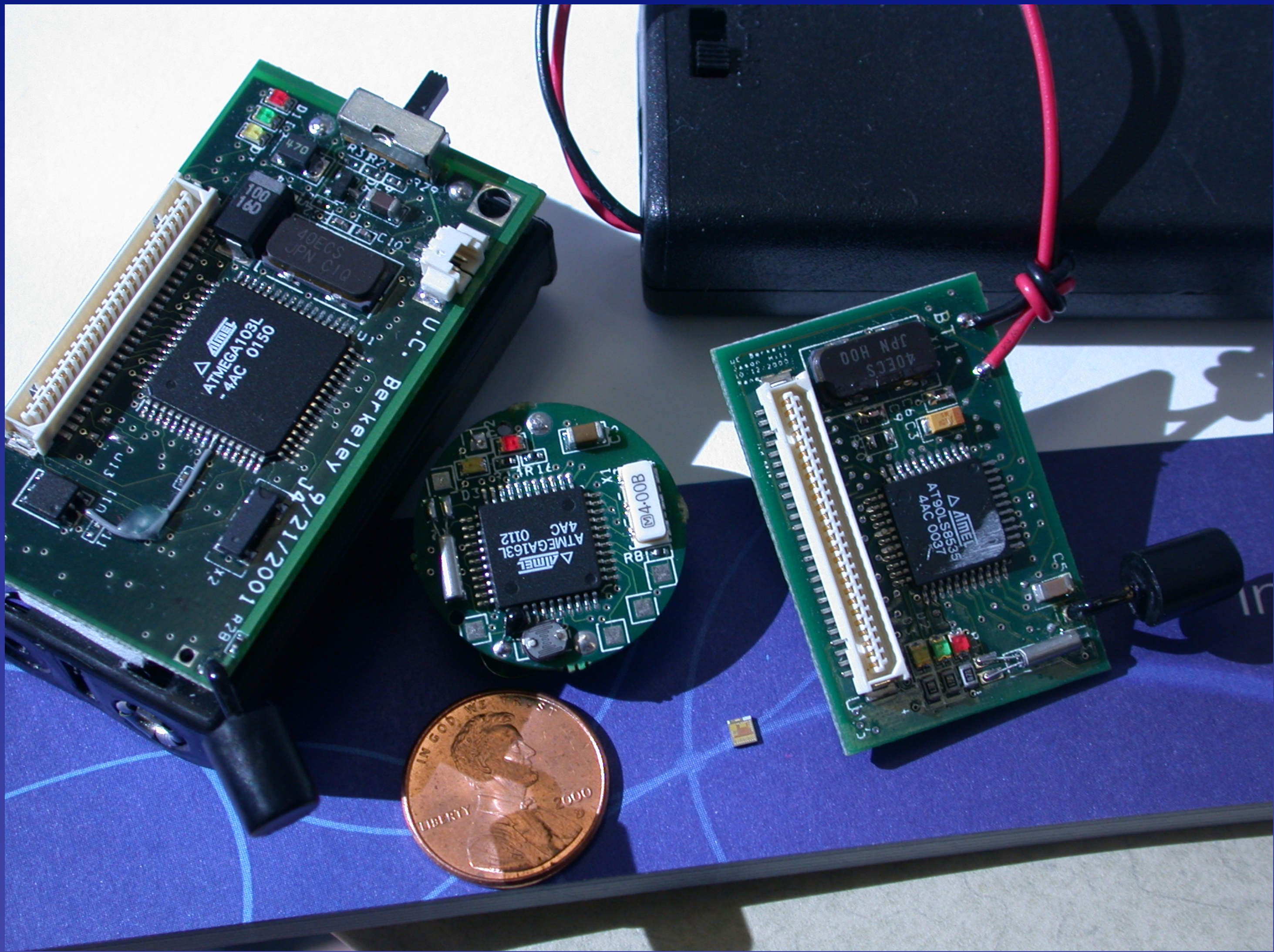
Philip Levis and David Culler

UC Berkeley

Intel Research: Berkeley

A Sensor Network

Sensor Network Motes



A Sensor Network

A Sensor Network

Bottom Line

- Need in-situ programming
- Has to be:
  - Small
  - Expressive
  - Concise
  - Resilient
  - Efficient
  - Tailorable
  - Simple

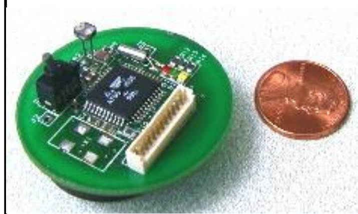
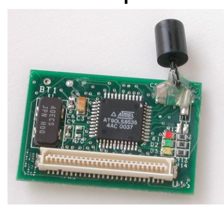
Proposal

Maté: A Tiny Virtual Machine for Sensor Networks

Outline

- Sensor networks
- Requirements
- Maté
- Evaluation
- Conclusion

Technological Constraints

Mote Type	WeC	Rene	Rene2	Dot	Mica
					
Date	Sep-99	Oct-00	Jun-01	Aug-01	Feb-02
Microcontroller (4MHz)					
Type	AT90LS8535		ATMega163		ATMega103/128
Prog. Mem. (KB)	8		16		128
RAM (KB)	0.5		1		4
Communication					
Radio	RFM TR1000				
Rate (Kbps)	10				10/40
Modulation Type	OOK				OOK/ASK

Example Application Scenario

- Monitor Storm Petrel nesting on Great Duck Island
- Inaccessible: 50 nodes in bird nests
- Simple sense and send loop
- Runs every 8 seconds – low duty cycle
- Frequent reprogramming would be useful
  - Biologists don't know what they need until they see it!

Proposal: Use a Virtual Machine!

- Can express a wide range of applications
- Abstraction of complex operations
- Safe execution environment
- Interpretation overhead small
- Customizable instruction sets
- VM can handle code dissemination

System Requirements

Requirement	Maté provides
• Small	• 7286B code, 603B RAM
• Expressive	• Bytecode interpeter
• Concise	• GDI app is 19 bytes
• Resilient	• Safe execution environment
• Efficient	• Small CPU overhead
• Tailorable	• User-definable instructions
• Simple	• Viral self-programming

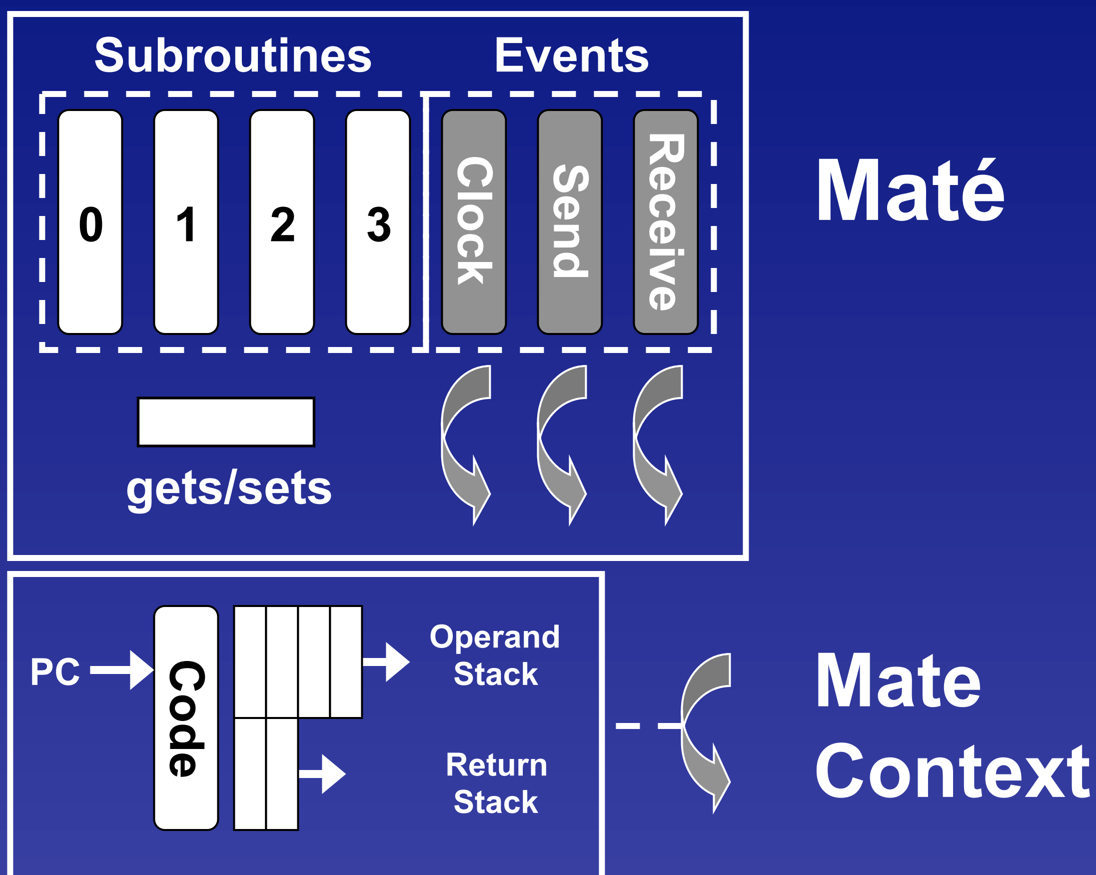
Why We Need a New VM

- Communication centric
- Extensibility
- Power a critical consideration
- JVMs (KVM, PicoJava, etc.)
  - Need over 50 KB of RAM
  - Strings? Are you crazy?
- FORTH
  - How do you install code?
  - Maté draws on FORTH's design decisions

Maté in a Nutshell

- Built on TinyOS, runs on rene and mica
- Three concurrent execution contexts
- Execution triggered by predefined events
- Two stack architecture
- Tiny code capsules self-propagate
- Communication and sensing instructions
  - built-in multihop routing

Maté Architecture



Maté Instructions

- One byte per instruction
- Three classes: basic, s-type, x-type
  - basic: data, arithmetic, communication, sensing
  - s-type: message headers
  - x-type: embedded operands (e.g. push constant)
- `usr0-7` instructions: tailorability

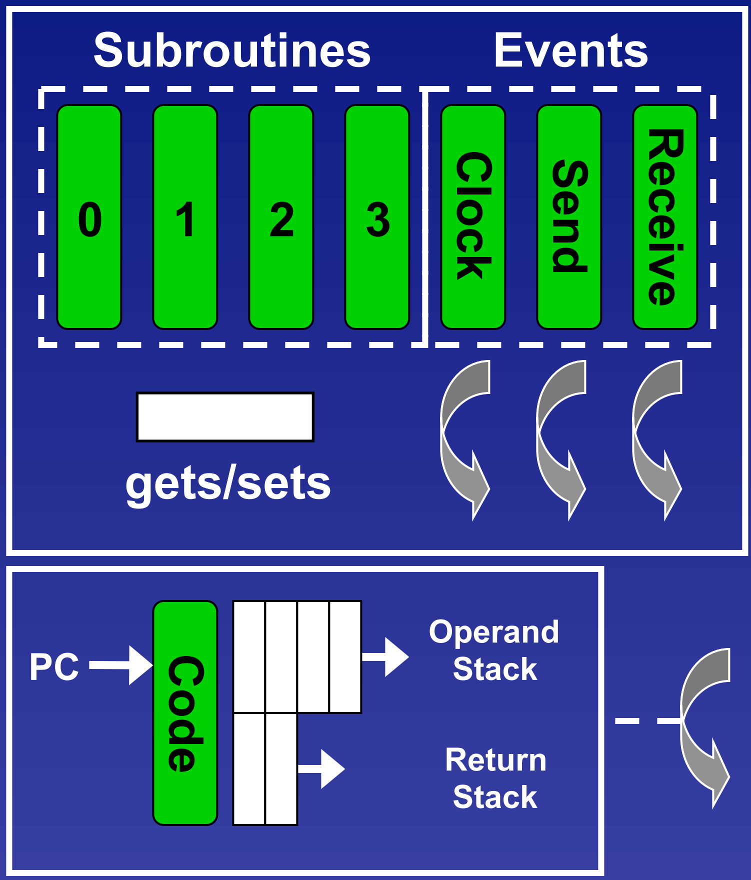
basic	`00iiiiii`	`i` = instruction
s-type	`01iiixxx`	`x` = argument
x-type	`1ixxxxxx`	

Maté Sense and Send

```
pushc 1    # Light is sensor 1
sense      # Push light reading on stack
pushm      # Push message buffer on stack
clear      # Clear message buffer
add        # Append reading to buffer
send       # Send message on built-in
halt       # ad-hoc protocol
```

Maté Capsules

- Hold up to 24 instructions
- Small enough to fit in a single TinyOS packet
  - atomic installation
  - no buffering
- Four types: send, receive, clock, subroutine
  - context-specific: send, receive, clock
  - shared: subroutines 0-3 (`call`, `ret`)



But, How Do Capsules Get There?

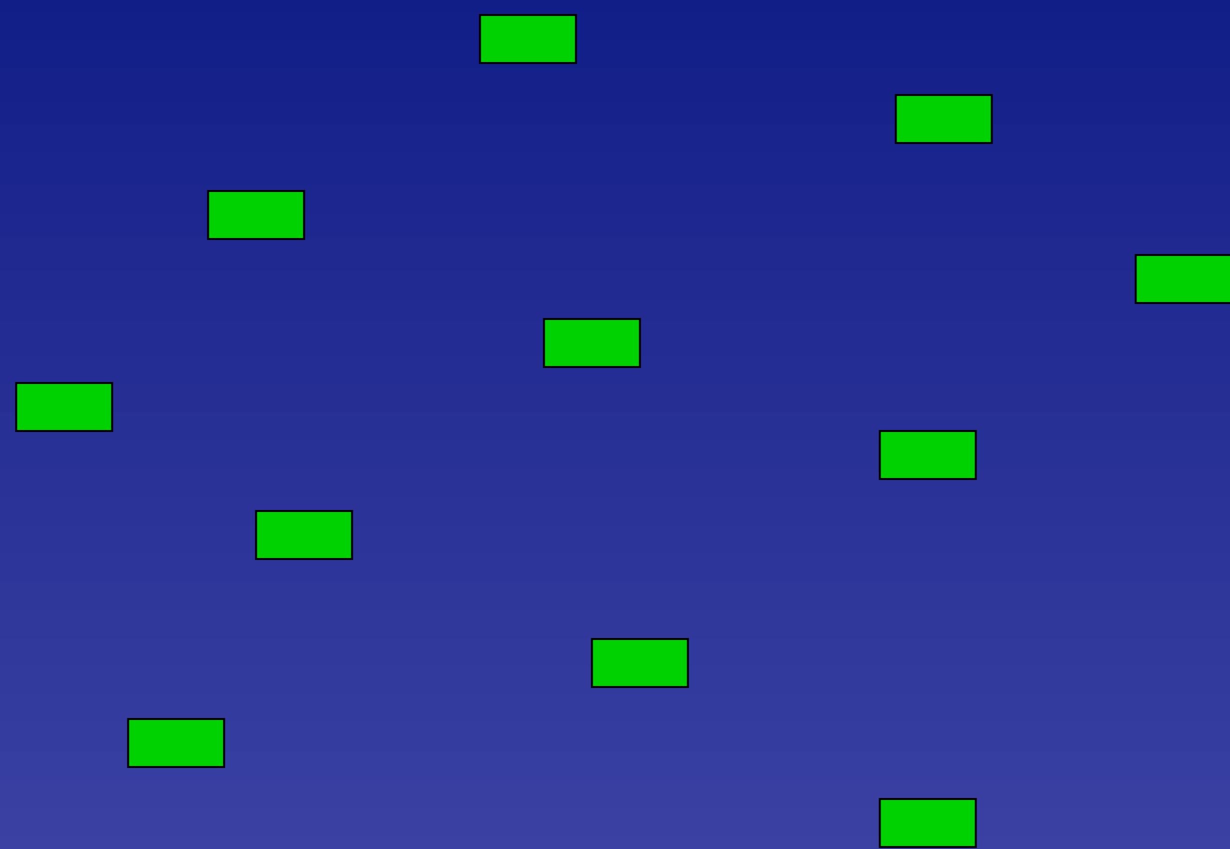
Viral Code

- Every capsule contains a version number
- Maté installs newer capsules it hears
- Programs can forward capsules
  - local broadcast
  - `forw`, `forwo`

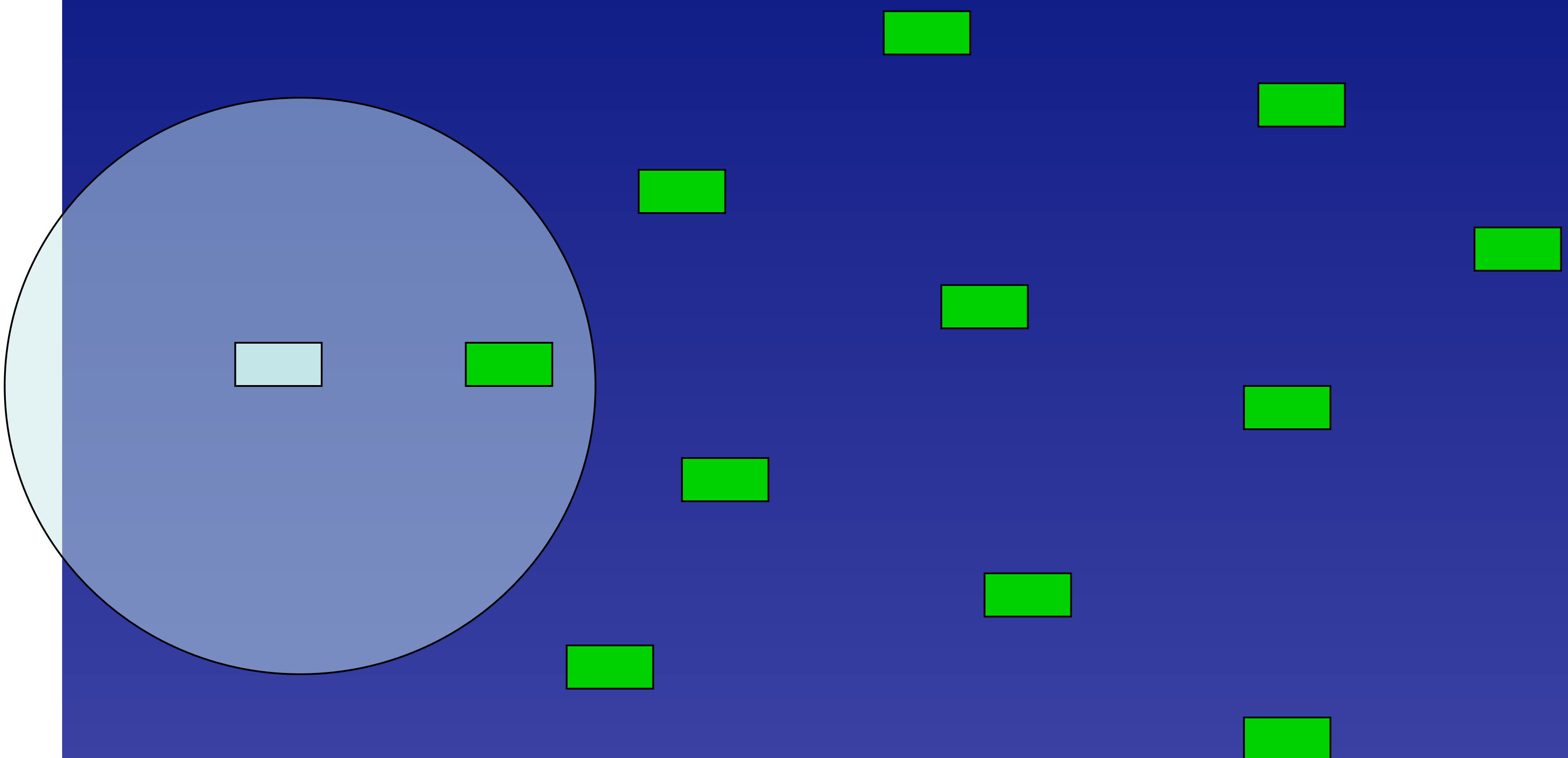
Self-Forwarding Sense and Send

```
pushc 1   # Light is sensor 1
sense     # Push light reading on stack
pushm     # Push message buffer on stack
clear     # Clear message buffer
add       # Append reading to buffer
send      # Send message
forw      # Forward this capsule
halt
```

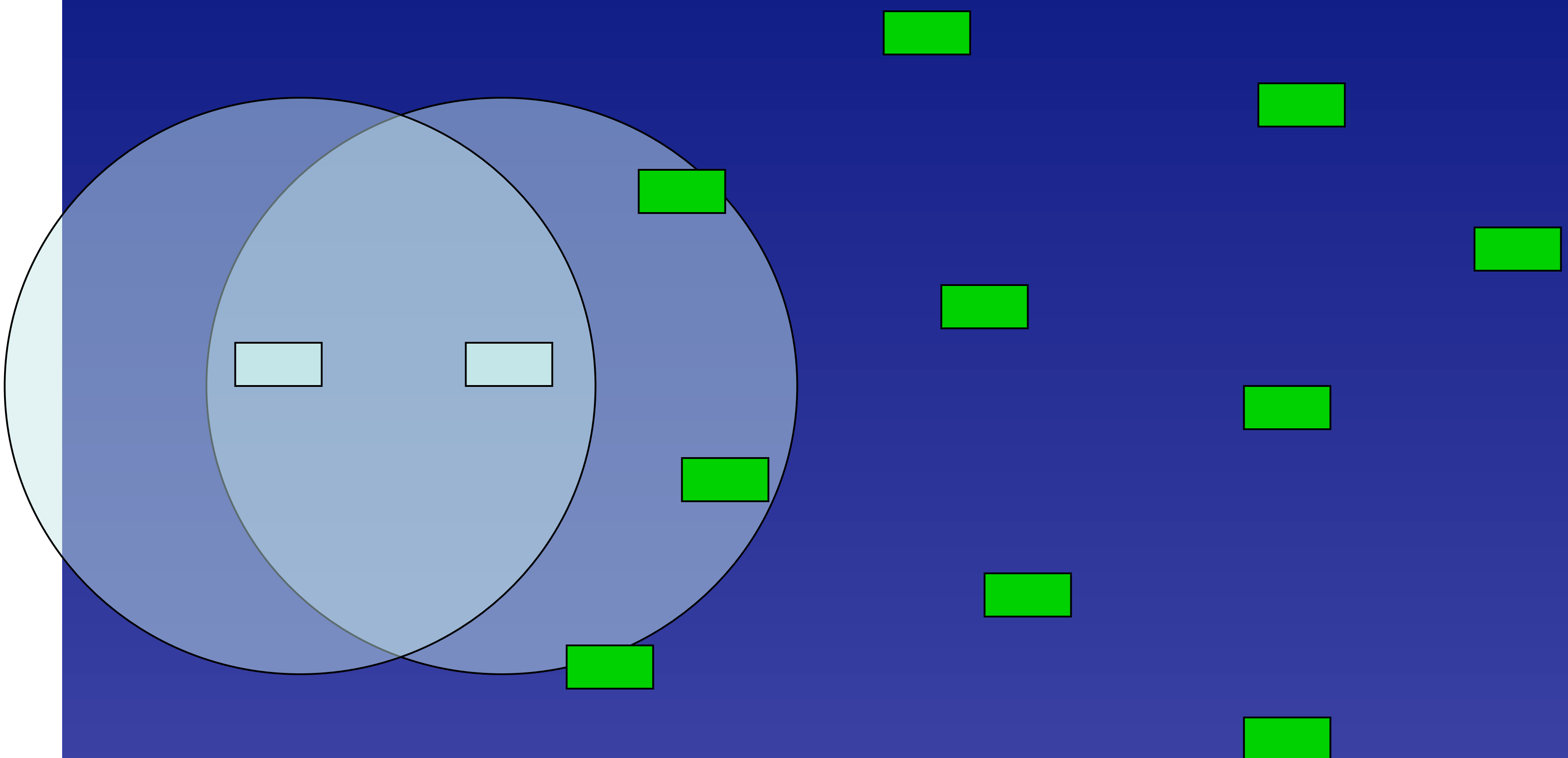
Propagation Example



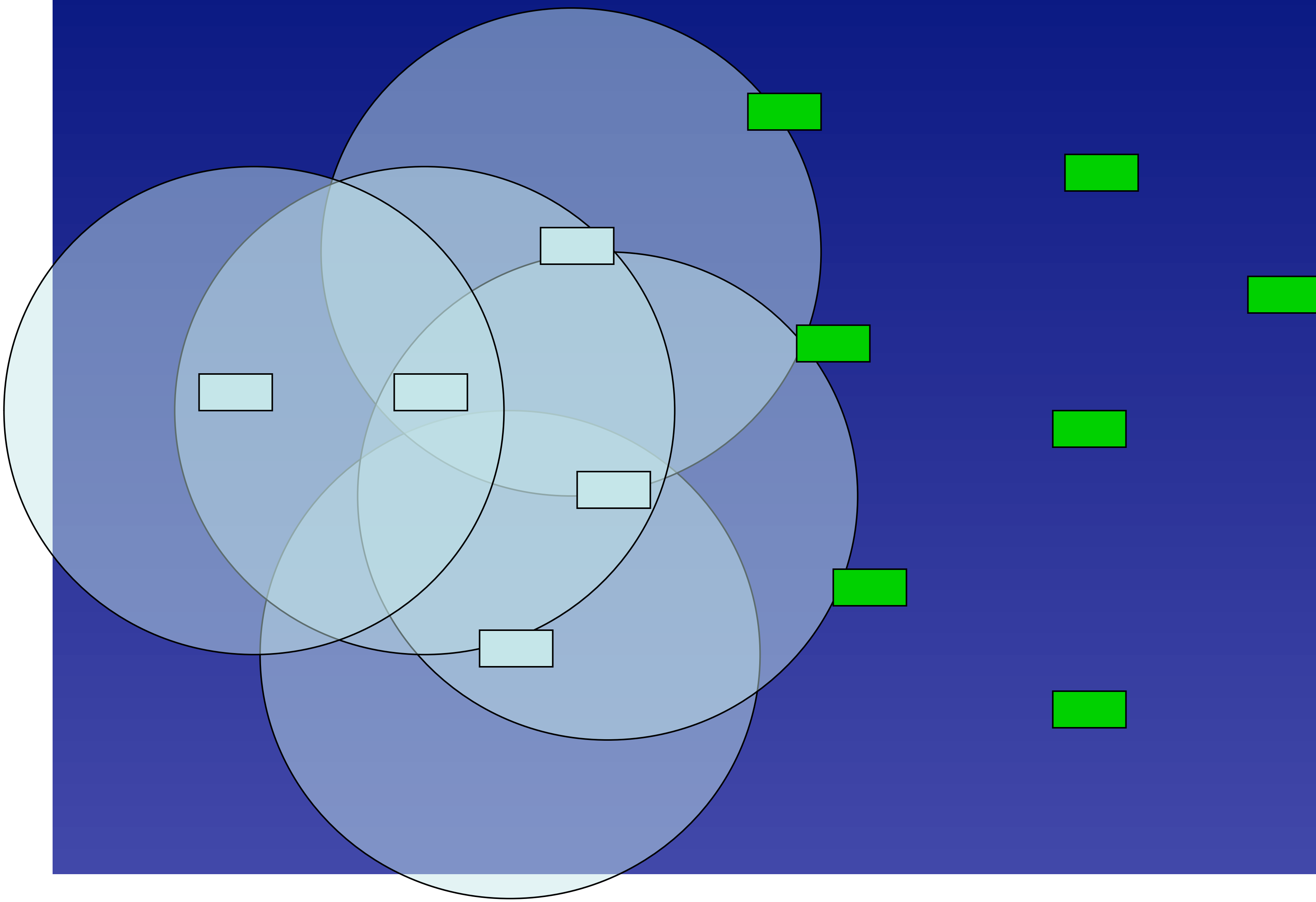
Propagation Example



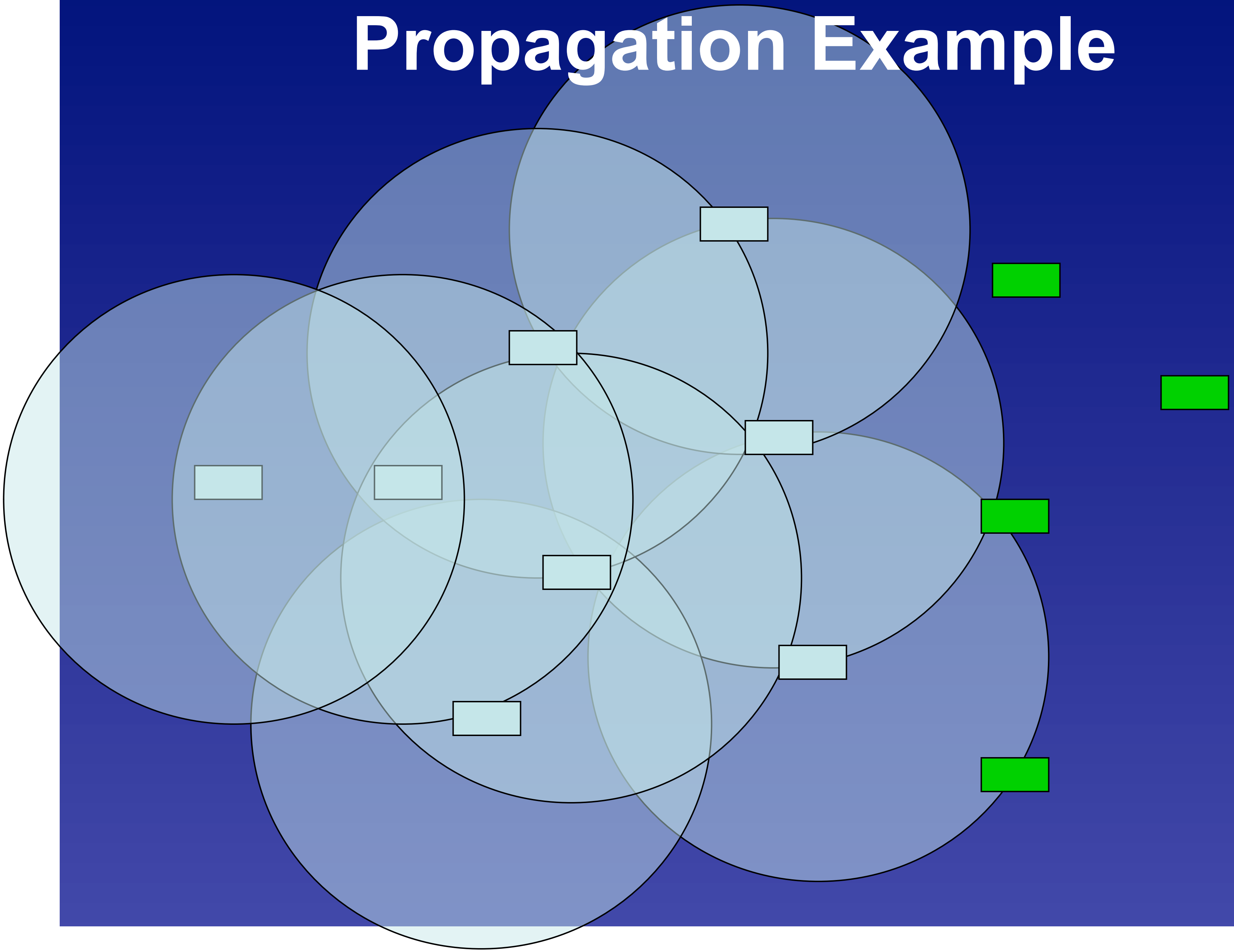
Propagation Example



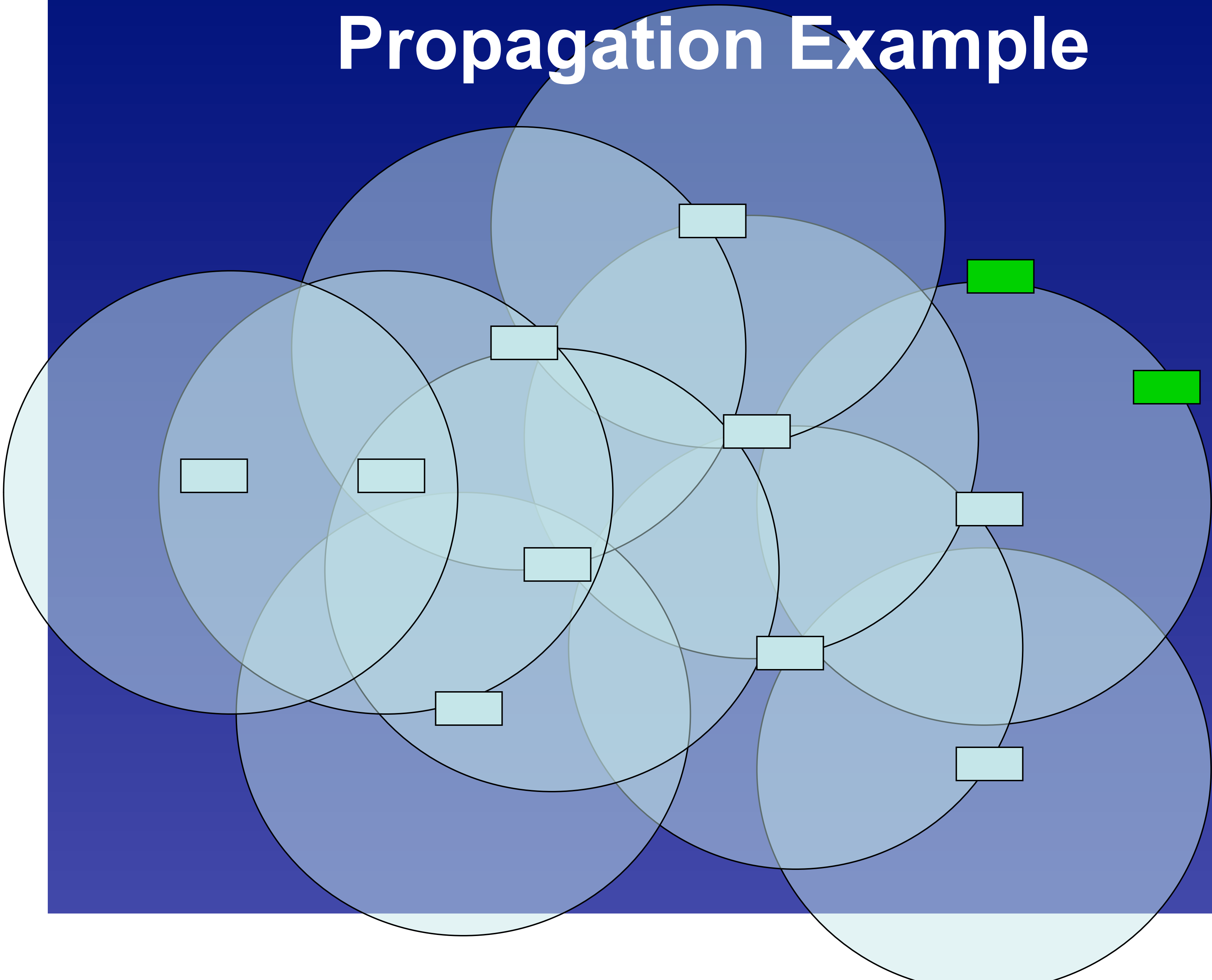
Propagation Example



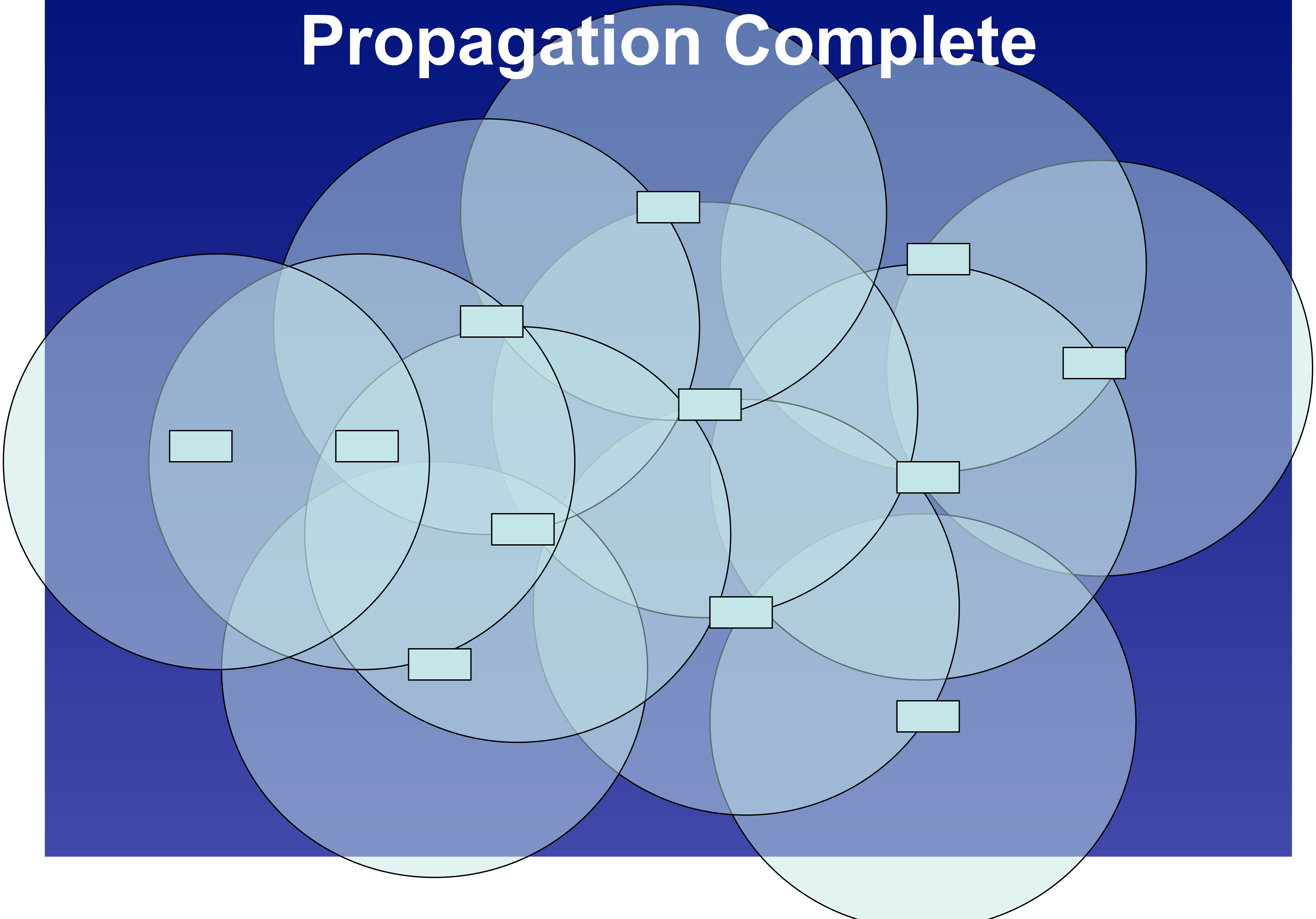
Propagation Example



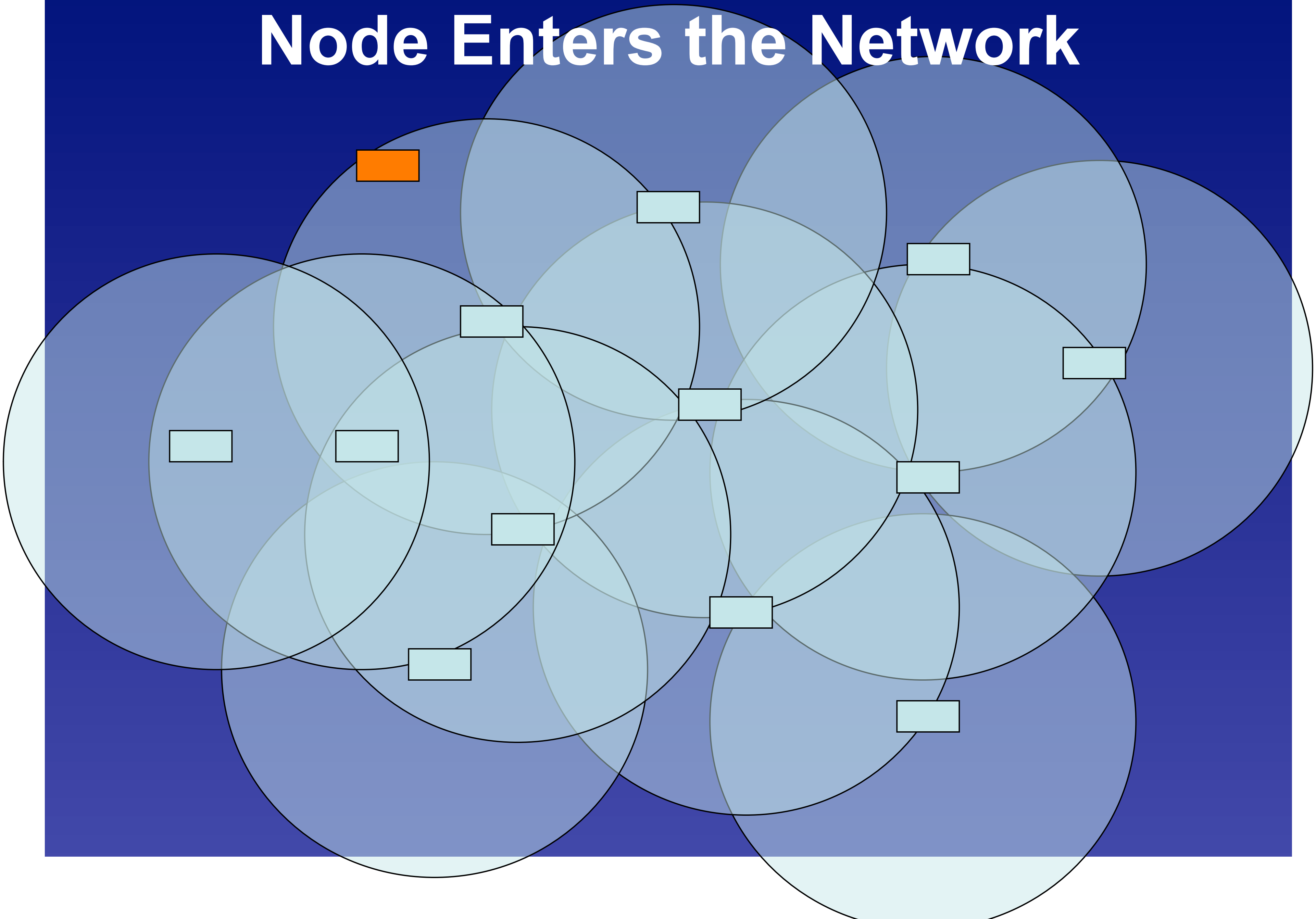
Propagation Example



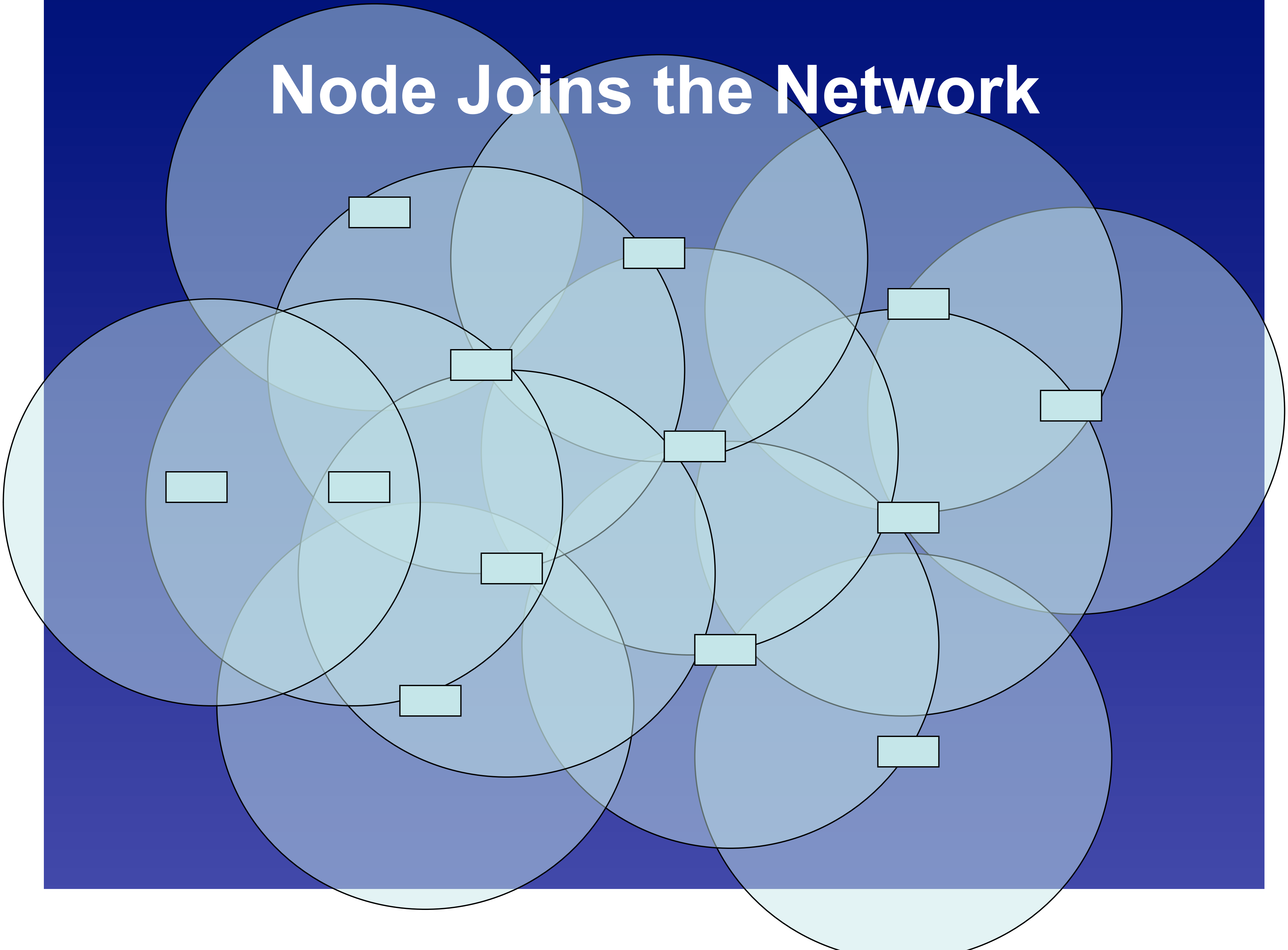
Propagation Complete



Node Enters the Network



Node Joins the Network



Evaluation

- What do we care about?
  - CPU cycles
  - bandwidth
  - *energy*
- Execution rate
- Code propagation behavior

Maté Interpretation Overhead

- ~10,000 instructions per second
- 34:1 to 1.03:1 compared to native code

Operation	Maté	Native	Cost
`and`	469 inst	14 inst	34:1
`rand`	435	45	9.5:1
`sense`	1342	396	3.4:1
`send`	685 + ~20,000	~20,000	1.03:1

Where Do the Cycles Go?

Instruction	Time	Time portion
`pushc 1`	40 us	0.06 %
`sense`	240 us	0.24 %
`pushm`	40 us	0.06 %
`clear`	40 us	0.06 %
`add`	50 us	0.08 %
`send`	60,000 us	99.44 %
`halt`	40 us	0.06%

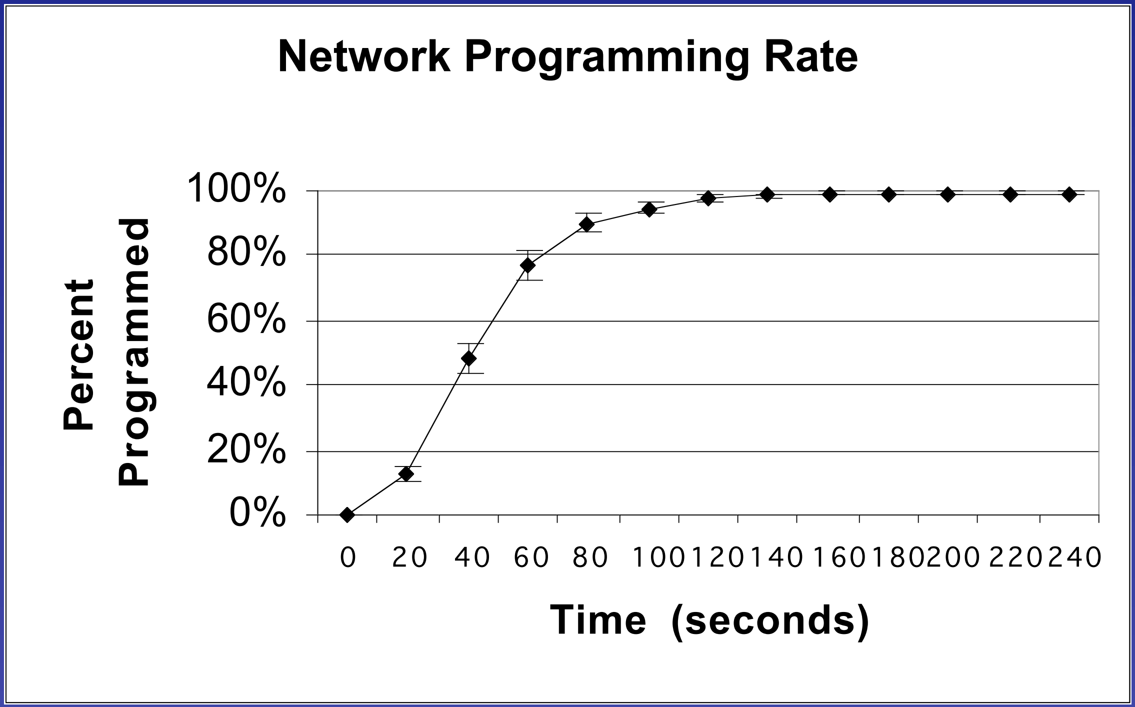
- Dominated by send
- Aggregate overhead: ~1.15:1

Code Propagation Methodology

- 42 node network
- 3 x 14 grid, spaced 20 cm apart
- 3 hop network (radio at very low power)
  - Cells were 15-30 nodes
- TinyOS 0.61 10Kb networking stack

Time to Complete Infection

- Self-forwarding timer capsule runs every 20 seconds
- Measures a quiet network (< 10% bandwidth)



Propagation Rate Scalability

- Timer capsule ran every second
- Capsule had a forwarding probability:
  - `if ((rand & 0x1) == 0x1) forward();`
- Network cell bandwidth: 16 packets/second

Probability	Expected Interval	Time
12.5%	8 s	23 s
25%	4 s	10 s
50%	2 s	21 s
100%	1 s	400 s

Energy Consumption

- **Maté imposes a CPU overhead**
- **Maté provides a reprogramming savings**

- **Rough energy cost comparison** (1 hop)
  - **full active: ~15mA x 3V x seconds**
  - **sense and send overhead/sample (2.5 ms)**
  - **sleep (~15 uA)**
  - **reprogramming savings (120 seconds)**

  ⇒ **50,000 samples equals one reprogram budget**

  ⇒ **400,000 seconds, 5 days**

Conclusions

- Maté can conserve energy
- Spectrum of reprogramming emerges
  - hardware
  - native code
  - bytecode interpreter

Future Work

- VM-land can replace user-land
- Higher-level languages: motlle
- Concurrency control
- Code propagation
- Bombilla: application specific Maté flavors

TinyOS Sense and Send

```
event result_t Timer.fired() {
  if (state == IDLE && call Photo.sense()) {state = SENSE;}
  return SUCCESS;
}

event result_t Photo.dataReady(uint16_t data) {
  if (state == SENSE) {
    packet->reading = data;
    if (call SendMsg.send(packet, sizeof(DataBuf)) {
      state = SENDING;
    } else {state = IDLE;}
  }
  return SUCCESS;
}

event result_t SendMsg.sendDone(TOS_MsgPtr msg) {
  if (state == SENDING) {state = IDLE;}
  return SUCCESS;
}
```