CAPITULO 2

Estado del arte de los DDOS

En este capítulo presentaremos brevemente el contexto actual en el que se encuentra el estudio de los ataques de denegación de servicio distribuido o DDOS. Primero presentaremos el esquema típico utilizado en este tipo de ataques así como las principales herramientas utilizadas por los atacantes.

A continuación realizaremos un rápido repaso de los diferentes grupos de trabajo así como de las distintas propuestas que se han realizado para conseguir la detección y minimización de los ataques DDOS.

Obviamente un entorno real es el único fiable al cien por cien para la comprobación o refutación de teorías, algoritmos y comportamientos. Sin embargo, en nuestro contexto (DDOS) resulta del todo imposible realizar las pruebas en real, ya que nuestro ámbito de estudio es toda Internet.

Una vez expuestas las limitaciones existentes en las soluciones propuestas en la actualidad, pasaremos a ubicar el nuevo ámbito de estudio de las redes de ordenadores: los simuladores de redes.

Describiremos brevemente los distintos simuladores que se han contemplado para el estudio sobre ataques DDOS así como el candidato seleccionado para el trabajo de la tesis.

Finalmente esbozaremos el cambio de los ataques DOS/DDOS y hacia dónde pueden la evolucionar en un plazo corto o medio.

2.1 Ataques de denegación de servicio distribuido

Tal y como ya se ha comentado, este tipo de ataques se basan en coordinar una serie de nodos conectados a Internet de forma que concentren su ataque simultáneamente y desde diferentes lugares sobre un mismo objetivo.

Pese a que los distintos ataques DDOS se centran en aspectos distintos, por ejemplo colapsar un servicio simulando la conexión de miles de usuarios a la vez o degradando el tiempo de respuesta generando tráfico espurio, el gran consumo del ancho de banda es su característica común.

Precisamente esta característica es la que les convierte en un inmenso peligro. No sólo pueden focalizar su ataque en cualquier nodo conectado a Internet, sino que colateralmente producen una ralentización del tiempo de respuesta para todos los

usuarios. No debemos olvidar que los paquetes de distintas comunicaciones comparten las mismas rutas.

Por un lado tenemos que las tecnologías ofrecen cada vez conexiones más rápidas y con mayor ancho de banda disponible. Por otro lado tenemos el constante crecimiento de nodos conectados a Internet, lo que nos conduce a que cada vez se utilice un mayor número de nodos en este tipo de ataques.

Obviamente puede darse el caso que de forma voluntaria y consciente varios centenares de usuarios se coordinen en un ataque concreto. Sin embargo, la realidad documentada nos enseña que generalmente los ordenadores implicados en un ataque lo son de forma involuntaria. Por otro lado, si un atacante utiliza su propio ordenador, tarde o temprano será localizado y sancionado por los organismos competentes.

De esta forma, uno o varios atacantes (*hackers*) se dedican a buscar sistemas vulnerables (sin el último parche de Windows, o con versiones antiguas de programas o sistemas operativos) dónde instalan un programa que les permite obtener un control remoto del ordenador (ver figura 2-1).

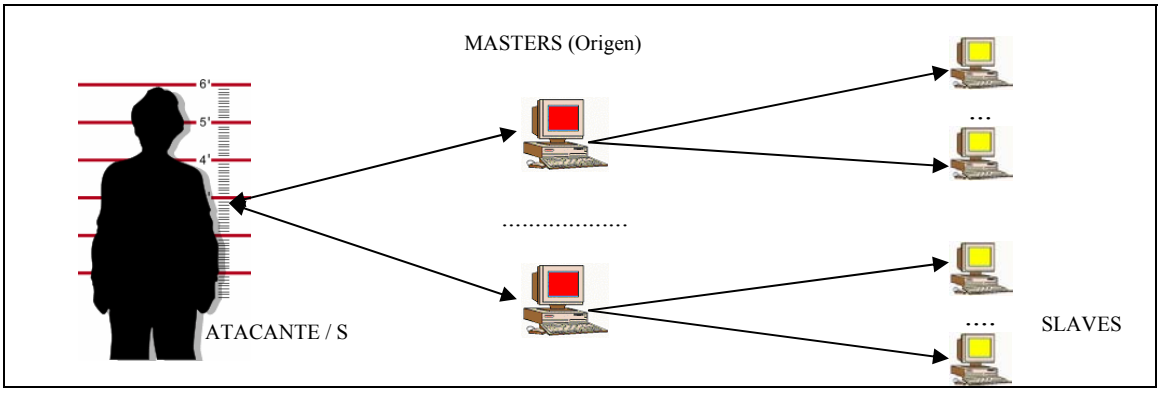

FIG. 2-1: Esquema de control utilizado en ataques DDOS.

En una red con millones de ordenadores conectados, la probabilidad de encontrar sistemas vulnerables o que nos permitan su uso ilícito aumenta enormemente al socializarse las conexiones permanentes. Simplemente es cuestión de tiempo.

Una vez que el atacante o atacantes han conseguido un cierto número de ordenadores bajo su control, los gestionan formando una estructura jerárquica que les permite por un lado mantener su anonimato y por otro controlar un gran número de nodos de forma sencilla y eficiente. Para más información de los métodos utilizados consultar la bibliografía [Far96][VM01][MP03][Ver03].

El atacante o atacantes controlan de forma directa una serie de nodos denominados "*masters*" que a su vez replican las órdenes recibidas a los nodos "*slave*" que tienen a su cargo (ver figura 2-1).

Actualmente las herramientas más utilizadas para este tipo de ataques son [Ver03]:

- Tribe Flood Network / **TFN** [Dit99-3].
- El proyecto **TRINOO** [Dit99] también conocido como TRIN00.
- **SHAFT** [DDL00].
- **STACHELDRAHT** [Dit99-2]**,** basada en código del TFN.
- El **TFN2K** [BT00], revisión de la herramienta TFN.

Además, también se utilizan técnicas de ataque indirecto o reflexión (***reflection DDOS***) [Gib02][Lar03][MP03][MP03-1] consistentes en falsear la dirección de origen de los datagramas (ver figura 2-2).

La idea subyacente en este tipo de ataques es aprovechar varios cientos de ordenadores con una conexión "lenta" para lanzar masivamente peticiones falsas a servidores con un gran ancho de banda disponible.

De esta forma, al substituir la dirección real del origen por la del nodo que deseamos atacar logramos que todas las respuestas inunden al nodo atacado hasta colapsarlo mediante el consumo de todo su ancho de banda.

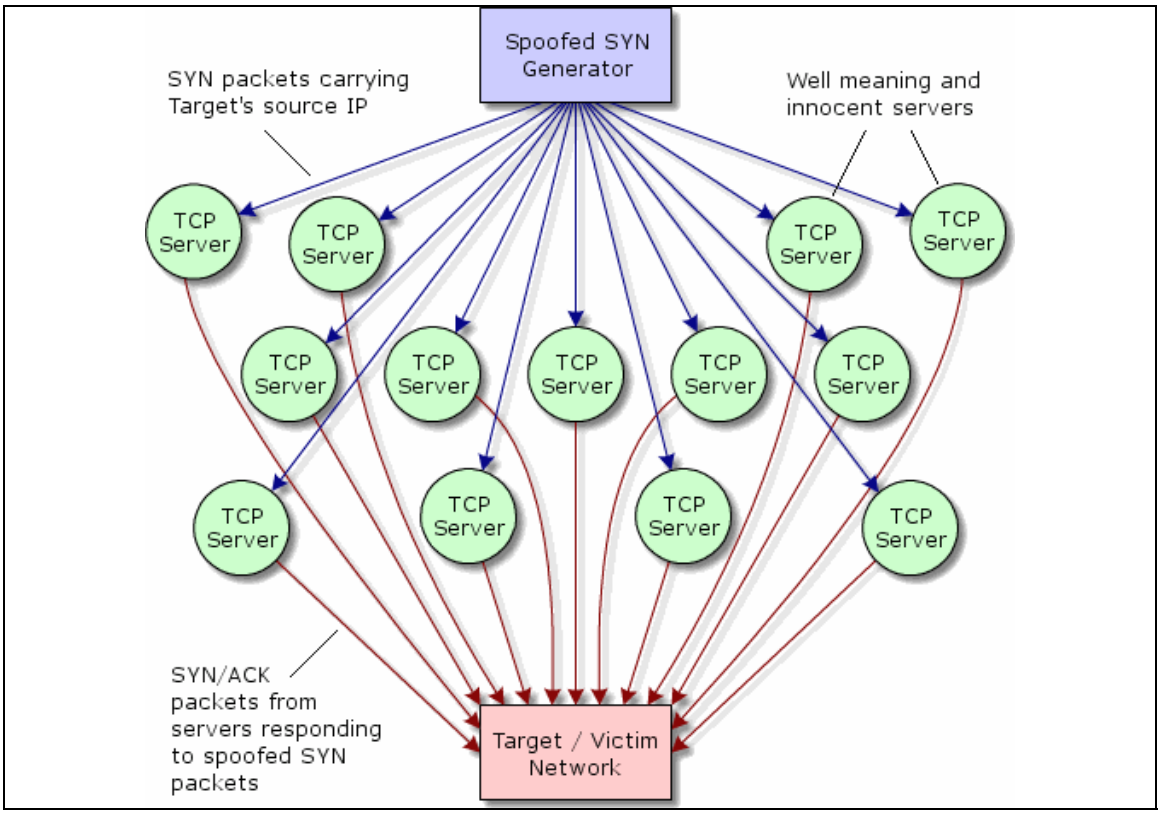

FIG. 2-2: Esquema del ataque DRDOS.

El último paso, una vez decidido el tipo de ataque a realizar, consiste en lanzar el ataque sincronizado y simultáneo sobre la víctima, que suele verse desbordada por completo mientras dura el ataque.

2.2 Respuestas actuales a los ataques DDOS

Una vez enumerados los principales ataques de denegación de servicio o DDOS realizaremos una breve introducción a los sistemas anti-DDOS existentes actualmente.

En un ataque de denegación de servicio podemos diferenciar de forma clara tres entidades distintas (ver figura 2-3):

1. **Sistemas de origen:** Hace referencia a los múltiples puntos dónde se origina la generación masiva de datagramas IP.

2. **Sistemas intermedios**: Son los sistemas pasivos por dónde circulan los paquetes IP hasta llegar a su destino.

3. **Sistema final:** Destino final del ataque.

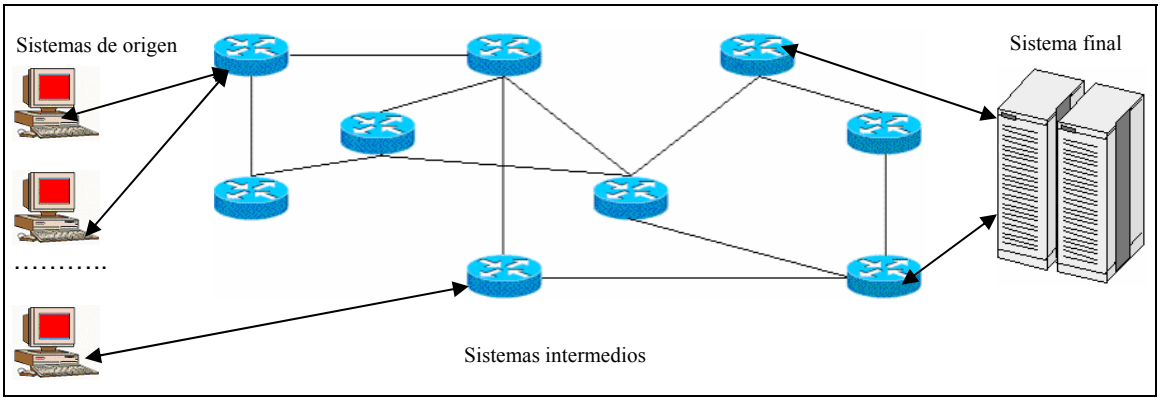

FIG. 2-3: Esquema de ataque DOS/DDOS.

Si analizamos el problema de la denegación de servicio en profundidad veremos que no hay recetas o soluciones mágicas que nos permitan evitar este tipo de ataques. Sin embargo, la bibliografía actual recoge cinco requisitos básicos que debería cumplir cualquier solución propuesta [MP03-2]:

1. Debemos utilizar una solución distribuida para solventar un problema distribuido. Las soluciones locales carecen de sentido puesto que no alcanzarán nunca la visión global que un ataque distribuido genera.

2. La solución propuesta no debe en ningún caso penalizar el tráfico de los usuarios legítimos. Muchas propuestas se basan en complejos sistemas de informes y filtros que mantienen listas y cuentas de todos los paquetes recibidos. Estos sistemas penalizan al global de los usuarios ralentizando el sistema.

3. Esta solución debe de ser robusta y universal. Debe ser válida para amenazas externas e internas. El sistema debe identificar[9R] los nodos y usuarios legítimos así como detectar y aislar si fuera necesario los nodos comprometidos o maliciosos.

4. El sistema propuesto debe de ser viable en su aplicación. La medida debe de ser adoptada por toda Internet, lo que implica que debe de ser sencilla y realizable con un coste razonable de recursos.

5. Debe de ser una solución incremental. Internet es un sistema heterogéneo dónde es imposible forzar la aplicación de un nuevo protocolo. El sistema propuesto debe permitir su aplicación gradual y ser compatible con el resto de los sistemas dónde aún no esté implementada.

La mayoría de las soluciones existentes en la actualidad están basadas en sistemas clásicos de defensa como los firewalls y sistemas de detección de Intrusos (IDS [Nor99][Car00][Des03] [Ver03]). Estos programas se encargan de filtrar todo el tráfico existente en nuestra red buscando mediante patrones y heurísticas indicios de actividad maliciosa para bloquearla.

Las recomendaciones de los fabricantes de equipos de comunicaciones como Cisco en su documento "*Defining strategies to project against TCP SYN denial of service attacks*" [WWW37] simplemente pueden recomendar acciones simples como aumentar el tamaño de la pila TCP, disminuir el tiempo de espera para el establecimiento de una conexión y filtrar el tráfico de salida para evitar el uso de direcciones IP falsas [Tan03].

Multops [GP01], Reverse Firewall [WWW154] o D-WARD [Obr01] son soluciones de este tipo que se encuentran instalados al final de la cadena de ataque (sistema atacado).

De esta forma, su eficacia es mínima debido a que si bien si que pueden detectar y bloquear la entrada de ataques DDOS a nuestra red, el consumo de ancho de banda se realiza igualmente, con lo que la respuesta a usuarios legítimos se ve interrumpida.

[9R] Por ejemplo mediante el uso de herramientas criptográficas.

En la actualidad empiezan a surgir soluciones agregadas o distribuidas como el ACC (*Aggregated-based Congestion Control*) [Mah02], SOS (*Secure Overlay Service*) [KMR02] o DefCOM (*Defensive Cooperative Overlay Mash*) [MP03-2].

Sin embargo estas soluciones requieren de comunicación directa con el resto de sistemas intermedios, lo que significa que también deben estar instalados en el resto de los sistemas intermedios para realizar un trabajo cooperativo.

Además, tal y como se ha indicado anteriormente debido al carácter mundial de Internet (múltiples países con distintas leyes, distinto hardware…) esto aún no se ha conseguido y parece una tarea destinada al fracaso.

En la realidad, esta necesidad de implantar globalmente la solución limita su eficacia y expansión únicamente a las redes de investigación, ya que al no dar una solución gradual ni estar aceptados como standards las diferentes propuestas, casi ningún sistema existente en producción los implementa.

2.3 Simuladores de redes de ordenadores

Debido al ámbito de nuestro estudio, nos encontramos con que nuestro campo de trabajo es toda Internet. Obviamente resulta del todo imposible el hecho de plantearse realizar las distintas pruebas en el dominio real de nuestro estudio. Es más, incluso nos resulta muy difícil el intentar realizar pruebas reales dentro de una misma universidad.

Al igual que ha pasado en otros campos de la ciencia, la creación de modelos de comportamiento de las redes y el aumento de potencia de los ordenadores ha permitido la creación de programas de simulación.

Igualmente y aun suponiendo perfectos los programas de simulación, resulta del todo impensable modelar Internet para la realización de cualquier tipo de experimento.

Por otro lado en la actualidad nadie conoce exactamente la topología existente ya que varía diariamente con el número de nodos conectados a Internet. Además las conexiones sin hilos (*wireless*) complican aún más el conocimiento de la distribución real de Internet.

Sin embargo, la simulación de pequeñas partes puede ser de gran utilidad, ya que el hecho de que Internet sea una red IP puede permitirnos verla como la suma de millones de subredes IP más pequeñas.

Como parte de nuestro estudio hemos realizado diferentes pruebas entre los programas de simulación existentes para determinar cual de ellos es el más idóneo para nuestro estudio.

Cabe destacar que para la realización de este estudio y debido a la limitación de recursos existente, todos los programas utilizados cumplen los requisitos de software libre, gratuito y exento de patentes.

Los resultados obtenidos en las evaluaciones de los programas de simulación presentados a continuación han sido realizados en la siguiente plataforma [WWW35]:

- Un ordenador Dual Pentium III a 600Mhz.
- 256MB de RAM.
- Sistema Operativo Linux con kernel 2.4.8
- J2SDK-1.3

En cuanto al esquema de red seleccionado para comprobar las capacidades de modelación de los programas analizados podemos observarlo en la figura 2-4.

Para este estudio se ha escogido un típico sistema formado por dos redes locales de ordenadores a 10Mbps. Estas redes se encuentran conectadas entre sí mediante una conexión a 1.5Mbps de los dos routers.

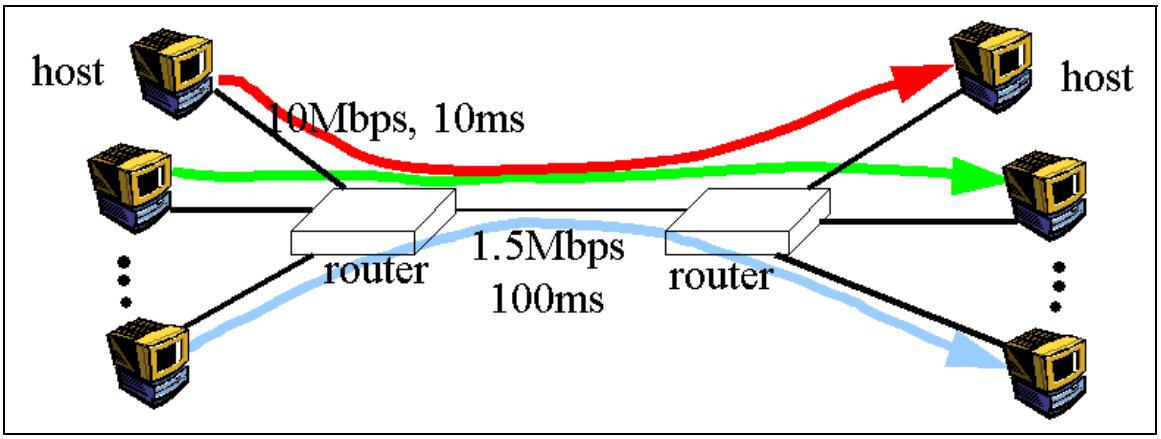

FIG. 2-4: Esquema de ataque DOS/DDOS.

2.3.1 NEtwork Simulation Testbed (NEST)

Este simulador desarrollado en la universidad norteamericana de Columbia (USA) es un programa de propósito general que permite la modelación de sistemas distribuidos de redes así como de protocolos básicos [WWW29].

La versión analizada es la 2.6 + patch 3.

Este sencillo simulador escrito en lenguaje C proporciona un entorno gráfico simple y sustenta su arquitectura en un modelo cliente/servidor.

Sus principales inconvenientes son que actualmente está sin mantenimiento, lo que limita muchísimo el soporte a los posibles usuarios, y que proporciona unas características de modelado excesivamente simples para nuestro estudio.

Por otro lado, la lentitud en la ejecución de las simulaciones desaconseja su uso para modelos medios o grandes.

2.3.2 Maryland Routing Simulator (MaRS)

El MARS es un simulador de redes desarrollado en el departamento de ciencias de la computación de la universidad de Maryland (USA) [WWW30]. Este programa nace como la evolución de otro simulador más antiguo denominado NetSim.

La versión analizada es la 2.1.

Este programa desarrollado en C dispone de dos interfaces, uno en modo simple y otro en modo gráfico desarrollado con las librerías gráficas X Motiv.

Su principal característica es la de proporcionar un entorno de simulación de algoritmos de rutado simple, lo que no permite el estudio de todo el abanico de posibilidades que necesitamos.

Por otro lado también carece de un soporte estable en la actualidad y no proporciona las capacidades de modelar comportamientos de sistemas autónomos o simulaciones en tiempo real.

2.3.3 REalistic And Large network simulator (REAL)

Este programa desarrollado en la de la universidad de Cornell (USA) [WWW31] se construyó a partir del simulador NEST dotándolo de varias mejoras para el estudio de la congestión de redes conmutadas. Entre sus características principales cabe destacar una mayor eficiencia en la simulación que su predecesor, lo que minimiza el tiempo de ejecución y un soporte nativo para la familia de protocolos TCP/IP.

La versión analizada es la 5.0.

Este programa no permite el estudio de sistemas o parámetros que no afecten de forma directa al flujo de conexiones TCP/IP principal. De esta forma, la capacidad de modelar un sistema real queda muy limitada.

Como sucede en los anteriores programas, el soporte ofrecido para este simulador es prácticamente inexistente en la actualidad.

2.3.4 Ns-2 (Network simulator 2)

El simulador **Ns-2** [WWW32] parte del código escrito para el simulador REAL con el objetivo de crear un simulador discreto de redes TCP. Incorpora las capacidades de rutado y multicast en redes de cable y satélite (*wireless connection*).

Este proyecto está patrocinado por DARPA, Xerox y el instituto de ciencias de la información (ISI) de la universidad del sur de california (USA).

La versión analizada es la 2.7.

Este simulador permite la modelación de las estructuras deseadas y actualmente ofrece un cierto soporte al usuario. Sus principales inconvenientes son la complejidad de las estructuras que utiliza (debido principalmente a la herencia de otros simuladores) y que permite únicamente la simulación discreta de eventos, lo que elimina la posibilidad de simulaciones continuas en tiempo real.

2.3.5 S3 project / Scalable Simulation Framework

Este proyecto de simulación creado por la colaboración de DARPA, Institute for security technology studies at Dartmouth y Renesis Corporation es uno de los mas reputados actualmente [WWW33][WWW34].

Este completo programa de simulación creado en C++ proporciona dos interfaces (APIs) de programación en los lenguajes JAVA y C++ así como un lenguaje específico de modelación denominado DML (*Domain Modeling languaje*).

Es altamente escalable e implementa masivamente el paralelismo permitiendo el uso de prácticamente todos los protocolos existentes en Internet de forma nativa (IP, UDP, TCP, OSPF, BGP…).

La versión analizada es la 2.0.

Su paradigma de simulación se basa en las cinco clases que proporciona al usuario para modelar su sistema (*Entity, inChannel, outChannel, Process y Event)* y permite únicamente una simulación discreta. Por otro lado la interacción con la simulación que se realiza es únicamente a través del DML, lo que elimina una rápida interactividad entre el usuario y la simulación.

Este programa permite la obtención de un alto grado de flexibilidad que es necesario en nuestro estudio. Además actualmente existe un soporte para usuarios.

Como puntos negativos cabe destacar su falta de rendimiento en las versiones gratuitas, ya que la obtención del paquete optimizado es de pago (lo distribuye la empresa Renesys). Las versiones gratuitas de este simulador [WWW36] si ofrecen un cierto soporte al usuario todo y que no se acercan a la estabilidad y velocidad que proporciona el simulador de pago.

2.3.6 J-Sim (JAVA Simulator)

El J-Sim es uno de los últimos programas de simulación creados en la comunidad universitaria. Este proyecto nace con la colaboración de NSF, DARPA, CISCO, la universidad de Illinois (USA) y Ohio (USA).

Este simulador está creado en JAVA y TCL utilizando una arquitectura de componentes autónomos. De esta forma se proporciona el conjunto de clases básicas para su funcionamiento y se permite la extensión o creación de nuevas clases mediante JAVA por parte de los usuarios.

La versión analizada es la 1.3.

El control de la simulación se realiza mediante scripts/TCL de forma que existe una consola de trabajo dónde interactivamente podemos ir dando órdenes al simulador. Las simulaciones que soporta este programa son discreta y continua o tiempo real.

Al igual que SSFnet proporciona soporte para la gran mayoría de protocolos utilizados en Internet así como multicast y QOS (*Quality Of Service*). Desde el punto de vista de usuario la posibilidad de poder interaccionar con el simulador TCL facilita el trabajo ya que en SSFnet se ha de retocar el programa DML constantemente en un proceso iterativo.

2.3.7 Resultados

En este apartado presentaremos algunos de los resultados obtenidos mediante la realización de pruebas con los distintos programas de simulación [WWW35] utilizando como modelo el esquema de red de la figura 2-4.

El programa de simulación utilizado para la modelación del sistema en el simulador J-Sim se puede observar en la figura 2-5. Para la ejecución del programa se utiliza la siguiente sintaxis de llamada:

```
java OneBottleneck #connections simulationTime interval<nn> event radix
```

```
import drcl.comp.*;
import drcl.inet.*;
import drcl.inet.data.RTKey;
import drcl.inet.data.RTEntry;
import drcl.inet.transport.TCP;
import drcl.inet.transport.TCPSink;
import drcl.inet.core.PktDispatcher;

public class OneBottleneck implements Runnable
{
	static boolean EXIT = true;
	static Component root_ = null;
	static int nc_ = 0;
	static double simulationTime = 0.0;
	static double interval = 100.0;

	public static void main(String[] args)
	{
	  if (args == null || args.length < 2) {
			System.out.println("Usage: java OneBottleneck <#connections> "
			+ "<simultion_time> ?interval<nn>? ?event? ?radix?");
				return;
			}

			nc_ = Integer.parseInt(args[0]);
			simulationTime = Double.valueOf(args[1]).doubleValue();
			boolean event_ = false;
			boolean radix_ = false;
			boolean linkemu_ = true;

			if (args.length > 2) {
			    for (int i=2; i<args.length; i++)
					if (args[i].equals("event")) event_ = true;
					else if (args[i].equals("radix")) radix_ = true;
					      else if (args[i].startsWith("interval"))
							interval = Integer.parseInt(args[i].substring(8));
			}

			System.out.println("Constructing a network of" + (2*nc_+2) + " nodes with " + nc_ + " TCP
connections");
			if (event_) System.out.println("-- Event simulation --");
			if (radix_) {
				System.out.println("-- Radix tree implementation --");
				drcl.inet.core.RT.IMPLEMENTATION = drcl.inet.core.RT.RADIX_TREE;
			}
			if (linkemu_) System.out.println("-- Link emulation --");

			long time_         = System.currentTimeMillis();
			Node[] senders_    = new Node[nc_];
			Node[] receivers_  = new Node[nc_];
			Node[] routers_    = null;
			root_ = new Component("test");
			Component.Root.addComponent(root_);

			// senders and router0 take address space (0, addrDivider-1)
			// receivers and router1 take address space (addrDivider, 2*addrDivider-1)
			long addrDivider = 1;
			while (addrDivider < (nc_+1)) addrDivider<<=1;
				System.out.println("address divider = " + addrDivider);
			try {
				 System.out.print("Create topology...");

			// adjacency matrix
			int[][] adjMatrix_ = new int[2*nc_ + 2][];
			for (int i=0; i<nc_; i++)
				adjMatrix_[i+2] = new int[]{0};
			for (int i=0; i<nc_; i++)
				adjMatrix_[i+nc_+2] = new int[]{1};
			adjMatrix_[0] = new int[nc_+1];
			adjMatrix_[1] = new int[nc_+1];
			adjMatrix_[0][0] = 1;
			adjMatrix_[1][0] = 0;
			for (int i=0; i<nc_; i++) {
				adjMatrix_[0][i+1] = i+2;
				adjMatrix_[1][i+1] = i+nc_+2;
			}

			Link link_ = null;
			if (!linkemu_) {
				link_ = new Link();
				link_.setPropDelay(0.01);
			}
			InetUtil.createTopology(root_, adjMatrix_, link_, false);
			routers_ = new Node[]{(Node)root_.getComponent("n0"), (Node)root_.getComponent("n1")};
			for (int i=0; i<nc_; i++) {
				senders_[i] = (Node)root_.getComponent("h" + (i+2));
				senders_[i].addAddress(i);
				receivers_[i] = (Node)root_.getComponent("h" + (i+nc_+2));
				receivers_[i].addAddress(i + addrDivider);
			}
			routers_[0].addAddress(nc_);
			routers_[1].addAddress(addrDivider + nc_);
			System.out.print("\n\nBuilding...");
			NodeBuilder nb_ = new NodeBuilder(".nb");
			nb_.setBandwidth(10000000); // 10Mbps
			if (linkemu_) {
				nb_.setLinkEmulationEnabled(true);
				nb_.setLinkPropDelay(0.01);
			}
			nb_.build(routers_);
			nb_.build(senders_, "tcp drcl.inet.transport.TCP\nsource  -/tcp
drcl.inet.application.BulkSource");
			nb_.build(receivers_, "tcpsink drcl.inet.transport.TCPSink\nsink  -/tcpsink
drcl.inet.application.BulkSink");
```

```
                        routers_[0].setBufferSize(0, 4096 * nc_); // 4KB per connection
                        routers_[0].setBandwidth(0, 1500000); // 1.5Mbps
                        routers_[1].setBandwidth(0, 1500000); // 1.5Mbps
                        if (linkemu_)
                                ((CoreServiceLayer)routers_[0].getComponent("csl")).setLinkPropDelay(0, 0.1);
                        else
                                ((Link)routers_[0].getPort("0").getPeers()[0].getHost()).setPropDelay(0.1);

                        System.out.print("\n\nSetup routes...");

                        // router0
                        CoreServiceLayer csl_ = (CoreServiceLayer)routers_[0].getComponent("csl");
                        csl_.addRTEntry(new RTKey(0L, 0L, addrDivider, -addrDivider, 0, 0),
                                        new RTEntry(new drcl.data.BitSet(new int[]{0})), Double.NaN);
                        for (int i=0; i<nc_; i++)
                                csl_.addRTEntry(new RTKey(0L, 0L, i, -1L, 0, 0),
                                                new RTEntry(new drcl.data.BitSet(new int[]{i+1})), Double.NaN);

                        // router1
                        csl_ = (CoreServiceLayer)routers_[1].getComponent("csl");
                        csl_.addRTEntry(new RTKey(0L, 0L, 0L, -addrDivider, 0, 0),
                                        new RTEntry(new drcl.data.BitSet(new int[]{0})), Double.NaN);
                        for (int i=0; i<nc_; i++)
                                csl_.addRTEntry(new RTKey(0L, 0L, i+addrDivider, -1L, 0, 0),
                                                new RTEntry(new drcl.data.BitSet(new int[]{i+1})), Double.NaN);

                        System.out.print("\n\nSetup TCP...");

                        for (int i=0; i<nc_; i++) {
                                TCP tcp_ = (TCP)senders_[i].getComponent("tcp");
                                tcp_.setPeer(i+addrDivider);
                                tcp_.setMSS(950);
                                ((TCPSink)receivers_[i].getComponent("tcpsink")).setMSS(950);
                        }
                        }
                        catch (Exception e_) {
                                e_.printStackTrace();
                                System.out.println("\n" + e_);
                        }
                        catch (OutOfMemoryError e_) {
                                System.out.println("\n" + e_);
                        }

                        System.out.println("Time for building scenario = " + ((System.currentTimeMillis() - time_) /
1000.0));

                        System.out.println("Start simulation");
                        drcl.comp.ACARuntime sim_ = null;
                        if (event_)
                                sim_ = new drcl.sim.event.SESimulator();
                        else {
                                sim_ = new drcl.sim.process.SMMTSimulator();
                                ((ARuntime)sim_).setMaxWorkforce(1);
                        }
                        sim_.takeover(root_);
                        sim_.stop();
                        sim_.addRunnable(.001, new OneBottleneck(sim_, time_));
                        Util.operate(senders_, "start");
                        sim_.resume();
                }

                OneBottleneck ()
                {}

                drcl.comp.ACARuntime sim;
                long starttime;

                OneBottleneck (drcl.comp.ACARuntime sim_, long starttime_)
                {
                        sim = sim_;
                        starttime = starttime_;
                }

                public void run()
                {
                        double now_ = sim.getTime();
                        if (now_ < simulationTime) {
                                if (now_ < 10.0)
                                        sim.addRunnableAt(10.0, this);
                                else if (now_ < interval)
                                        sim.addRunnableAt(interval, this);
                                else
                                        sim.addRunnable(interval, this);
                                System.out.println("     " + sim.getTime() + "\t" +
(sim.getWallTimeElapsed()/1000.0));
                        }
                        else {
                                System.out.println("     " + sim.getTime() + "\t" +
(sim.getWallTimeElapsed()/1000.0));
                                System.out.println("Time for the whole thing = " + ((System.currentTimeMillis()
- starttime) / 1000.0));
                                System.out.println("Simulation ends.\n" + sim.diag());
                                System.out.println("Event Queue:\n" +
((drcl.util.queue.Queue)sim.getEventQueue()).diag(false));
                                if (EXIT) System.exit(0);
                                else sim.stop();
                        }
                }
}
```

FIG. 2-5: Programa para el simulador J-Sim.

Para el simulador Ns-2 se ha utilizado como programa de modelación el explicitado en la figura 2-6. Su sintaxis de ejecución es la siguiente:

```
ns2 ns2_test.tcl <#connections> <simulationTime>
```

```
# bottleneck scenario with various # of tcp flows

if {$argc < 2} {
          puts "usage: ns ns2_test.tcl <#connections> <Simulation_time>"
          exit 0
}

proc writetime currentTime_ {
          global wallTimeStart
          puts "\t$currentTime_\t\t[expr [clock seconds] - $wallTimeStart] seconds"
}

set nc [lindex $argv 0]
set time_ [lindex $argv 1]
#set time_ 50.0; # simulation time

proc finish {} {
          global ns_
          global wallTimeStart time_
          puts "\t$time_\t\t[expr [clock seconds] - $wallTimeStart] seconds"
          puts "[$ns_ info2]"
           exit 0
}

puts "---------- NC=$nc, time=$time_ ----------\n\tcreate nodes and connections"
set wallTimeStart [clock seconds]

set ns_ [new Simulator]
$ns_ rtproto Manual

set routers(0) [$ns_ node]
set routers(1) [$ns_ node]

$ns_ duplex-link $routers(0) $routers(1) 1.5Mb 100ms DropTail
$ns_ queue-limit $routers(0) $routers(1) [expr $nc * 4]
[$routers(0) get-module "Manual"] add-route-to-adj-node -default $routers(1)
[$routers(1) get-module "Manual"] add-route-to-adj-node -default $routers(0)

for {set i 0} {$i < $nc} {incr i} {
          set senders($i) [$ns_ node]
          $ns_ duplex-link $senders($i) $routers(0) 10Mb 10ms DropTail
          [$routers(0) get-module "Manual"] add-route-to-adj-node $senders($i)
          [$senders($i) get-module "Manual"] add-route-to-adj-node -default $routers(0)
          set receivers($i) [$ns_ node]
          $ns_ duplex-link $receivers($i) $routers(1) 10Mb 10ms DropTail
          [$routers(1) get-module "Manual"] add-route-to-adj-node $receivers($i)
          [$receivers($i) get-module "Manual"] add-route-to-adj-node -default $routers(1)

          # TCP and apps
          set agents_ [$ns_ create-connection-list TCP/Reno $senders($i) TCPSink $receivers($i) $i]
          set tcp_ [lindex $agents_ 0]

          set sink_ [lindex $agents_ 1]
          set source_($i) [$tcp_ attach-source FTP]
          $ns_ at 0.0 "$source_($i) start"
}

puts "\t\t\t[expr [clock seconds] - $wallTimeStart] seconds"

puts "\tSimulation Time"

set times ""
set index 0
for {set i 100} {$i <= $time_} {incr i 100} {
          append times " $i"
}
proc recursive_event {} {
          global index times ns_
          writetime [$ns_ now]
          $ns_ at [lindex $times $index] recursive_event
          incr index
}

$ns_ at $time_ "finish"
$ns_ at 10.0  "recursive_event"

set wallTimeStart [clock seconds]
$ns_ run $wallTimeStart
```

FIG. 2-6: Programa para el simulador Ns-2.

Finalmente a continuación presentamos el programa de simulación utilizado para la modelación del sistema en el simulador SSFnet en la figura 2-7. Para la ejecución del programa se utiliza la siguiente sintaxis de llamada:

```
tclsh gen.tcl <#connections>
```

```
if {$argc < 1} {
          puts {gen.tcl <#connections>}
          exit
}

set nc [lindex $argv 0]
#set simulationTime [lindex $argv 1]

set f [open "tmp.dml" w]

puts $f {

Net [

  frequency 1000000000    # 1 nanosecond time resolution

  # "left side" client can request data from any server
  # "right side" client can request data only from "left side" server

  traffic [
}

# clients:  3 - $nc+2
# servers:  $nc + 3 - 2*$nc+2

for {set i 0} {$i < $nc} {incr i} {
          set client_ [expr $i+3]
          set server_ [expr $nc+$i+3]

          puts $f [subst -nocommand {
    pattern [
      client $client_
      servers [nhi $server_\(0) port 1600]
    ]
          }]
}

set buffer_ [expr 4096*$nc]
puts $f [subst -nocommand {
  ]

  randomstream [
    generator "MersenneTwister"
    stream "seedstartingstring1234567890"
    reproducibility_level "host"
  ]

  # ROUTERS AND POINT-TO-POINT LINKS ----------------------------------

  router [
    idrange [from 1 to 2]
    graph [ProtocolSession [name ip use SSF.OS.IP]]
    interface [ id 0 buffer $buffer_ _extends .dictionary.T1]
    interface [ idrange [from 1 to $nc] _extends .dictionary.10BaseT]
    route [dest default interface 0]
  ]

  link [attach 1(0) attach 2(0) delay 0.1]
}]

for {set i 0} {$i < $nc} {incr i} {
          set client_ [expr $i+3]
          set server_ [expr $nc+$i+3]
          set if_ [expr $i+1]

          puts $f [subst -nocommand {
  link [attach 1($if_) attach $client_\(0) delay 0.01]
  link [attach 2($if_) attach $server_\(0) delay 0.01]
          }]
}

puts $f {
 # CLIENTS AND SERVERS ------------------------------------------------
}

for {set i 0} {$i < $nc} {incr i} {
          set client_ [expr $i+3]
          set server_ [expr $nc+$i+3]

          puts $f [subst -nocommand {
  host [ id $client_
    _extends .dictionary.standardClient
  ]
  host [ id $server_
```

```
    _extends .dictionary.standardServer
  ]
          }]
}

puts $f {
] # end of Net

# -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
dictionary [

  # standard network interfaces
  10BaseT [
    bitrate 10000000
    #latency 0.0001
    latency 0.0
  ]
  100BaseT [
    bitrate 100000000
    #latency 0.0001
    latency 0.0
  ]
  T1 [
    bitrate 1500000
    #latency 0.1
    latency 0.0
  ]

  standardClient [
    interface [id 0 _extends .dictionary.10BaseT]
    route [dest default interface 0]
    graph [
      ProtocolSession [
        name client use SSF.OS.TCP.test.tcpClient
        start_time 0.0            # earliest time to send request to server
        off_time 0.0              # to keep the connection busy all the time
        start_window 1.0          # send request to server at randomly chosen time
                                  # in interval [start_time, start_time+start_window]
        file_size 1000000000        # requested file size (payload bytes)
        _find .dictionary.appsession.request_size
        _find .dictionary.appsession.show_report
        _find .dictionary.appsession.debug
      ]
      ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
      ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
                       _find .dictionary.tcpinit]
      ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]
  standardClient_debug [
    interface [id 0 _extends .dictionary.10BaseT]
    route [dest default interface 0]
    graph [
      ProtocolSession [
        name client use SSF.OS.TCP.test.tcpClient
        start_time 0.0            # earliest time to send request to server
        off_time 0.0              # to keep the connection busy all the time
        start_window 1.0          # send request to server at randomly chosen time
                                  # in interval [start_time, start_time+start_window]
        file_size 1000000000        # requested file size (payload bytes)
        _find .dictionary.appsession.request_size
        _find .dictionary.appsession.show_report
        _find .dictionary.appsession.debug
      ]
      ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
      ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
                       _find .dictionary.tcpinit
                       _extends .dictionary.tcpdebug]
      ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]
  standardServer [
    interface [id 0 _extends .dictionary.10BaseT]
    route [dest default interface 0]
    graph [
    ProtocolSession [
      name server use SSF.OS.TCP.test.tcpServer
      port 1600           # server's well known port
      client_limit  10    # max number of contemporaneously allowed clients
                          # if omitted, default is no limit
      _find .dictionary.appsession.request_size
      _find .dictionary.appsession.show_report
      _find .dictionary.appsession.debug
      ]
      ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
      ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
                       _find .dictionary.tcpinit]
      ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]

  standardServer_debug [
    interface [id 0 _extends .dictionary.10BaseT]
    route [dest default interface 0]
    graph [
    ProtocolSession [
      name server use SSF.OS.TCP.test.tcpServer
      port 1600           # server's well known port
      client_limit  10    # max number of contemporaneously allowed clients
                          # if omitted, default is no limit
      _find .dictionary.appsession.request_size
      _find .dictionary.appsession.show_report
      _find .dictionary.appsession.debug
      ]
```

```
      ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
      ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
                       _find .dictionary.tcpinit
                       _extends .dictionary.tcpdebug]
      ProtocolSession [name ip use SSF.OS.IP]
    ]
  ]

  tcpinit[
    ISS 10000               # initial sequence number
    MSS 960                 # maximum segment size
    RcvWndSize  32          # receive buffer size
    SendWndSize 32          # maximum send window size
    SendBufferSize 128      # send buffer size
    MaxRexmitTimes 12       # maximum retransmission times before drop
    TCP_SLOW_INTERVAL 0.5   # granularity of TCP slow timer
    TCP_FAST_INTERVAL 0.2   # granularity of TCP fast(delay-ack) timer
    MSL 60.0                # maximum segment lifetime
    MaxIdleTime 600.0       # maximum idle time for drop a connection
    delayed_ack false       # delayed ack option
    fast_recovery true      # implement fast recovery algorithm
    show_report false        # print a summary connection report
  ]

  tcpdebug [
                             debug true
                             #rttdump rtt
                             #cwnddump cwnd
                             #rexdump rex
                             eventdump event
  ]

  # the trivial client-server protocol

  appsession [
    request_size  64        # client request datagram size (in bytes)
    show_report false        # print client-server session summary report
    debug false             # print verbose client/server diagnostics
  ]

] # end of dictionary
}

close $f
```

FIG. 2-7: Programa para el simulador SSFnet.

En la figura 2-8 podemos observar los distintos resultados obtenidos con los programas analizados al variar el parámetro de las conexiones realizadas de 0 a 10.000.

En la primera imagen podemos observar como en períodos cortos de simulación (100 segundos) correspondientes a 2.500 conexiones aproximadamente, SSFnet muestra ligeramente un mejor ratio numero de conexiones vs. tiempo.

Al continuar la simulación y alargar el número de conexiones a 10.000 podemos observar en la segunda imagen de la figura 2-8 la mejor evolución del simulador J-Sim.

Ns-2 presenta una saturación a parir de los 600 segundos (5.000 conexiones) y SSFnet da problemas al superar las 7.500 conexiones simuladas. Este comportamiento puede ser debido a un bug del motor empleado en las pruebas (*Out of memory exception*).

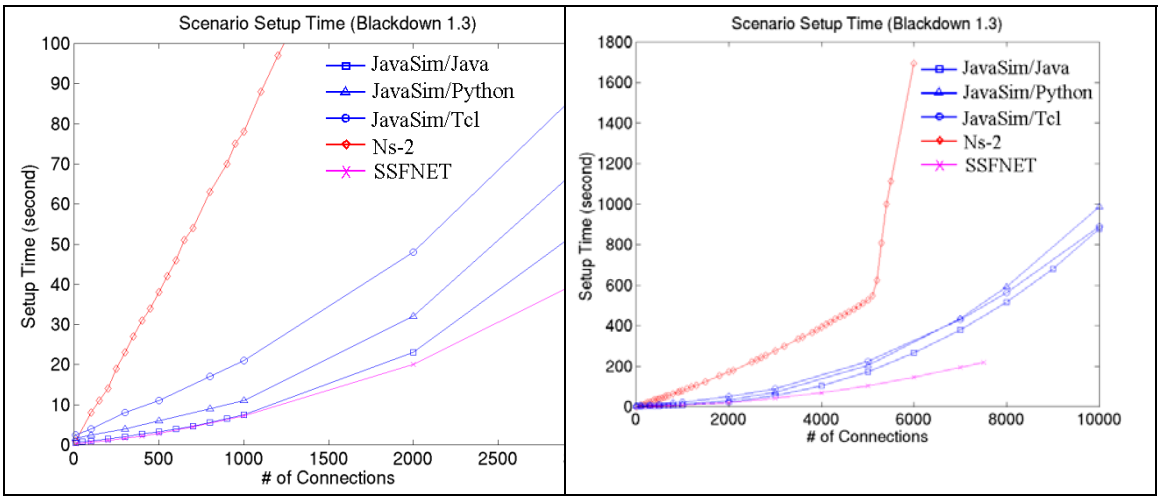

FIG. 2-8: Análisis de simulación de conexiones.

J-Sim presenta una escalabilidad mantenida durante toda la simulación llegando sin problemas a las 10.000 conexiones. De esta forma podemos observar que pese a dar un comportamiento ligeramente inferior a SSFnet hasta las 7.500 conexiones, nos permite la simulación de sistemas más poblados con una mejor estabilidad.

En la figura 2-9 podemos observar la comparación de los modos de simulación de eventos o simulación real (real time) de J-Sim con los demás programas.

En la primera imagen podemos ver como para la simulación de 5000 conexiones el comportamiento de los simuladores Ns-2 y SSFnet empieza a dispararse respecto el J-Sim. Una vez más se podría atribuir este comportamiento a un bug de las versiones analizadas ya que su comportamiento para pocas conexiones es excelente y extrañamente presenta errores al superar las 3.500 conexiones en SSFnet y las 4.500 en Ns-2.

Al extender la simulación a 10.000 conexiones podemos comprobar como la simulación en tiempo real (real time) tiene un ligero overhead respecto a la basada en eventos.
En cualquier caso el consumo de memoria y recursos de J-Sim permite el aumento de las conexiones en las simulaciones de forma estable, cosa que nos hace decantar nuestra opción de estudio hacia este programa.

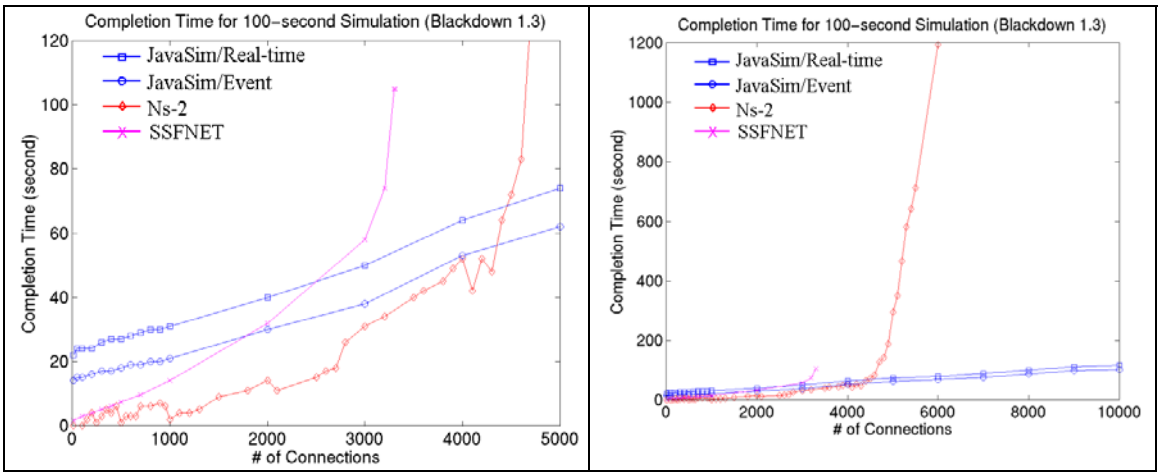

FIG. 2-9: Análisis de simulación de tiempo.

2.4 El futuro de los ataques DDOS

Una vez estudiado el contexto histórico de los ataques de denegación de servicio y su evolución hasta los ataques distribuidos actuales (MyDoom) podemos observar que:

- La frecuencia de su aparición es cada vez mayor. Si en un principio únicamente se realizaban estos ataques a grandes empresas (Yahoo, Ebay, Microsoft…) con el objetivo de ganar una cierta notoriedad, ahora cualquier punto de conexión (empresas, universidades, organismos oficiales…) es blanco indiscriminado de este tipo de ataques (SCO, Telefónica, IANA…).

  Las motivaciones han cambiado y en la actualidad se producen incluso casos de extorsión [Ley04] mediante la amenaza de DDOS a servicio de Internet como casinos o bancos.

- Los ataques se muestran cada vez más virulentos. De esta forma se ha pasado de conseguir el cese de un servicio de forma puntual unos minutos u horas a borrar la presencia de una empresa (Mydoom ante SCO) durante más de una semana.

- La sofisticación de los métodos utilizados ha evolucionado desde los sistemas semi-artesanales tipo *master/slave* dónde el ataque era controlado remotamente hasta sistemas automáticos dónde el ataque se lleva a cabo por cualquier ordenador infectado por un virus.

De esta forma, y visto el resultado de los trabajos realizados en el área de la prevención de DDOS hasta la fecha podemos extrapolar que el problema lejos de desaparecer o minimizarse crecerá siendo uno de los grandes caballo de batalla para la supervivencia de Internet.

Nuestra tesis versará sobre cómo atacar esta problemática minimizando en el daño que estos ataques puedan producir así como conseguir su detección y desactivación de forma efectiva.

2.4.1 Última vulnerabilidad en el protocolo TCP

El 20 de abril de 2004 se hace pública por los organismos oficiales [WWW42] [WWW43] una última vulnerabilidad en el diseño del protocolo TCP. Esta vulnerabilidad permite la finalización por parte de un tercero de una conexión TCP legítima, lo que en la práctica es un ataque de denegación de servicio.

El protocolo TCP [RFC793] proporciona una conexión fiable entre dos puntos cualesquiera conectados a Internet. Para el establecimiento de la conexión se utiliza el denominado protocolo de tres pasos (*three way handshake*) cuyo funcionamiento básico es el siguiente:

1. El cliente (ordenador que desea iniciar la comunicación) selecciona un número aleatorio de secuencia (***x*** en la figura 2-10). A continuación activa el *flag* de SYN en el campo de opciones y finalmente envía un segmento TCP al ordenador destino.

2. El servidor (ordenador con el que se quiere establecer una comunicación) recibe la petición y almacena el número de secuencia ***x***. Elige un número aleatorio (***y*** en la figura 2-10) que utilizará como número de secuencia y activa los *flags* SYN y ACK.

   Finalmente envía un segmento con el número de secuencia elegido y con una confirmación del valor recibido mas uno (***ACK x+1*** en la figura 2-10).

3. El cliente almacena el número de secuencia (***y*** en la figura 2-10). Activa el *flag* de ACK y finalmente envía una confirmación del número recibido mas uno ***(ACK y+1*** en la figura 1-14).

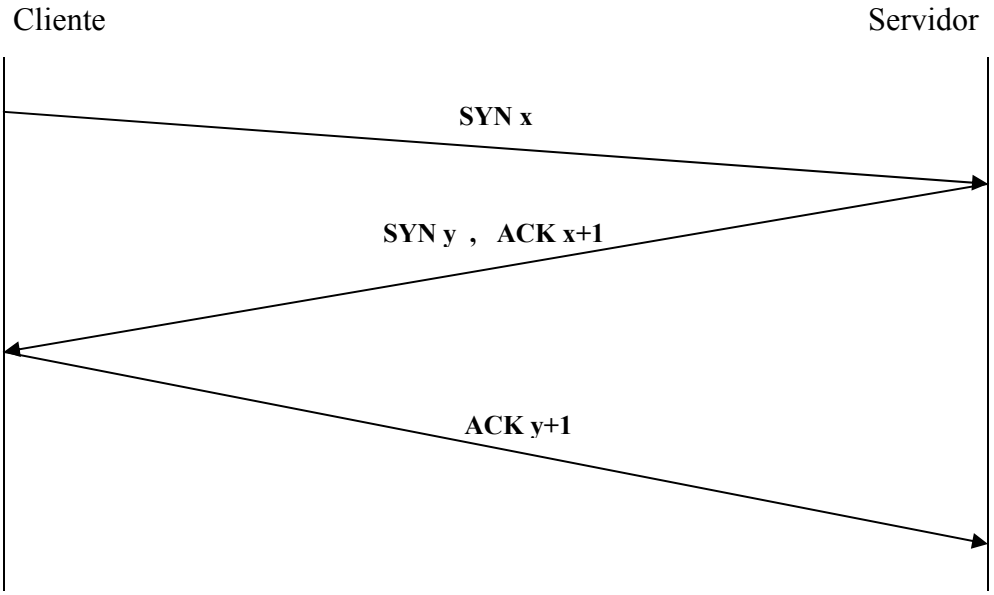

FIG. 2-10: Establecimiento de una conexión TCP (*three way handshake*).

Entre los parámetros que se negocian en el establecimiento de la conexión figura el tamaño en bytes de la ventana (*window*) de la conexión. Este parámetro permite la regulación de la comunicación y la mejora del rendimiento entre origen y destino, ya que en vez de enviar una respuesta por cada paquete recibido la envía por cada "ventana".

El ataque dado a conocer se basa en la posibilidad de que un tercero pueda enviar un paquete con el flag reset activado de forma que la conexión se cierre inmediatamente.

Una conexión TCP queda delimitada por una 4-tupla que contiene la dirección IP de origen, el número de puerto origen, la dirección IP de destino y el número de puerto destino. Para que un tercero pueda cerrar una conexión TCP que no le pertenece debe enviar un paquete con las direcciones IP adecuadas, los números de puertos correctos y el flag de RST (reset) activado [Wat04].

El número de secuencia es un campo de 32 bits [Ric98-1][RFC793] con lo que la probabilidad de acertar este número de secuencia suponiendo una distribución aleatoria es de 1/2^32 (0,00000000023283064365386962890625). Sin embargo según la especificación del protocolo TCP se aceptará cualquier número de secuencia que se encuentre en el rango de la ventana (*window*) definida previamente al iniciarse la conexión (*three way handshake*).

Debido a esta característica en su definición ampliamos enormemente las posibilidades de enviar un número de secuencia que se encuentre entre los permitidos por la ventana definida, ya que en lugar de enviarlos todos (1, 2, 3…) únicamente enviaremos uno por cada posible rango de la ventana (ver figura 2-11).

La conexión aceptará números de secuencia comprendidos entre el número de secuencia actual (SQN) y la ventana definida en el establecimiento de la conexión (W).

1. Suponemos una ventana de 1 byte.

Números de secuencia posibles (SQN+W): 0,1,2,3,4,5…., 2^32-1

Total de números de secuencia a probar: 2^32

2. Suponemos una ventana de 65535 bytes.

Números de secuencia posibles (SQN+W): 0, 65535, 131070, 196605, 262142,…, 4294901760

Total de números de secuencia a probar: 2^32 / 65535 = **65535!!!**

FIG. 2-11: Obtención del número de secuencia.

El descubrimiento de las direcciones IP de origen y destino no supone ningún problema ya que por un lado conocemos la dirección destino, la del servicio que queremos atacar, y por otro la dirección origen a la cual queremos evitar su acceso al servicio. Además programas de sniffing de red pueden simplificar más aún esta tarea [WWW44].

En cuanto al descubrimiento de los puertos de origen y destino, la realidad de los estudios [WWW42][WWW43][Wat04] demuestra que los sistemas operativos actuales utilizan sistemas predecibles. En el peor de los casos tenemos un número de 16 bits (2^16 = 65535) lo que nos da una probabilidad de 1/65535 (0,0000152587890625).

Este tipo de ataque es especialmente peligroso en conexiones TCP de larga duración que permiten la existencia del tiempo suficiente para que el atacante consiga enviar el datagrama adecuado.

Los servicios más afectados actualmente son los del protocolo de intercambio de rutas entre routers BGP (*Border gateway protocol*) y el servicio de nombres o DNS.

Paul Watson [Wat04] demuestra que en con una conexión xDSL y un tamaño de ventana de 65535 bytes es posible conseguir un paquete de estas características cada 5 minutos. En el caso de una conexión T1 (1.54Mbps) el tiempo se reduce a 15 segundos.

Obviamente en nuestro contexto de ataques de servicio distribuido (DDOS) podemos observar que con un número relativamente pequeño de máquinas, 10 líneas ADSL básicas suman 1.28Mbps, podemos conseguir este efecto prácticamente de forma inmediata.