Una Arquitectura Multiagentes para Argumentación Distribuida

Diego R. García Alejandro J. García Guillermo R. Simari

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253, (8000) Bahía Blanca, Argentina
{drg,ajg,grs}@cs.uns.edu.ar

La finalidad de este proyecto es diseñar e implementar un sistema multiagentes que permita implementar en forma distribuida un sistema de Argumentación Rebatible [5, 1, 2, 3, 6, 4]. En la Argumentación Rebatible las conclusiones de un agente son sustentadas con *argumentos*. Una conclusión se dice *garantizada* cuando el argumento que la sustenta no posee *derrotadores* (contra-argumentos que lo derrotan) o todos sus derrotadores son a su vez derrotados. Dado un argumento $\mathcal{A}$ pueden existir uno o varios derrotadores para $\mathcal{A}$. Cada derrotador ataca a un punto interno diferente del argumento el cual se denomina *punto de ataque*. La arquitectura aquí descripta puede utilizarse para implementar diferentes sistemas de argumentación rebatible que existen en la actualidad. En este trabajo se describe una implementación particular que utiliza DeLP (Defeasible Logic Programming) [4], un lenguaje que permite representar conocimiento con reglas rebatibles y estrictas, y las respuestas a las consultas son garantizas construyendo un *árbol de dialéctica*.

El sistema multiagentes estará formado por un agente Ag que construye un argumento para sustentar una conclusión c, y un conjunto no determinado de *agentes derrotadores* AD_i cuyo objetivo es construir argumentos derrotadores para atacar argumentos de otros agentes. Los agentes compartirán cierto conocimiento $\mathcal{K}$ que les permite construir argumentos y derrotadores. De esta forma, cuando se quiere decidir si una conclusión c está garantizada a partir del conocimiento $\mathcal{K}$, un agente Ag construirá un argumento $\mathcal{A}$ para sustentar c, y luego, agentes derrotadores intentarán construir contra-argumentos para derrotar al argumento $\mathcal{A}$ construido por Ag. Cada uno de estos agentes derrotadores se ejecuta independientemente del resto, y se concentrará en atacar al argumento $\mathcal{A}$ en un punto diferente. Como para derrotar a un argumento, cada uno de los agentes derrotadores AD_i debe encontrar un contra-argumento $\mathcal{C}$ que a su vez no sea derrotado, entonces el argumento $\mathcal{C}$ puede a su vez ser atacado por otros agentes derrotadores recursivamente.

La siguiente definición extraída de [4] muestra el proceso dialéctico de argumentación de DeLP que permite decidir si una conclusión está garantizada. Un argumento $\mathcal{A}$ para una conclusión h es denotado $\langle \mathcal{A}, h\rangle$. Un árbol de dialéctica para $\langle \mathcal{A}_0, h_0\rangle$, se denota $\mathcal{T}_{\langle \mathcal{A}_0, h_0\rangle}$, y se construye de la siguiente forma:

1. La raíz del árbol es etiquetada con $\langle \mathcal{A}_0, h_0\rangle$.
2. Sea N un nodo del árbol etiquetado $\langle \mathcal{A}_n, h_n\rangle$, y $[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \ldots, \langle \mathcal{A}_n, h_n\rangle]$ la secuencia de etiquetas del camino que va desde la raíz hasta el nodo N. Sean $\langle \mathcal{B}_1, q_1\rangle$, $\langle \mathcal{B}_2, q_2\rangle$, ..., $\langle \mathcal{B}_k, q_k\rangle$ todos los derrotadores de $\langle \mathcal{A}_n, h_n\rangle$ (cada uno para un punto de ataque diferente). Para cada derrotador $\langle \mathcal{B}_i, q_i\rangle$ $(1 \leq i \leq k)$, tal que, la línea de argumentación $[\langle \mathcal{A}_0, h_0\rangle, \langle \mathcal{A}_1, h_1\rangle, \langle \mathcal{A}_2, h_2\rangle, \ldots, \langle \mathcal{A}_n, h_n\rangle, \langle \mathcal{B}_i, q_i\rangle]$ sea aceptable, existe un nodo hijo N_i de N etiquetado con $\langle \mathcal{B}_i, q_i\rangle$. Si no existe ningún derrotador $\langle \mathcal{B}_i, q_i\rangle$ en tales condiciones, entonces el nodo N es una hoja.

Los nodos en un árbol de dialéctica se pueden marcar como "D" *derrotado*, o "U" *no derrotado*[1]. de la siguiente forma:

1. Todas las hojas de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ se marcan con "U" en $\mathcal{T}^*_{\langle \mathcal{A}, h \rangle}$.
2. Sea N un nodo interno de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$. El nodo N se marca con "U" si todo nodo hijo de N está marcado con "D", y N se marca con "D" si existe al menos un nodo hijo de N marcado con "U".

Una conclusión h sustentada por $\mathcal{A}$ está garantizada si la raiz de $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ es un nodo "U" (no derrotado).

La construcción del árbol de dialéctica y el proceso de marcado pueden realizarse en forma distribuida si, para un argumento dado, sus derrotadores son computados en paralelo. El modelo de argumentación distribuido que describimos aquí, está compuesto por un conjunto sitios, que son esencialmente servidores que reciben consultas de otros sitios y las procesan concurrentemente. El sistema multiagentes propuesto se ejecuta sobre estos sitios. Estos sitios pueden encontrarse en una misma computadora (host) o en diferentes máquinas, y se comunican entre si por medio de una red. Dentro de estos sitios se ejecutan concurrentemente los agentes derrotadores, que son creados como respuesta al pedido de un agente en otro sitio o en el mismo. De esta forma, los agentes estarán distribuídos en diferentes sitios, y cada uno de ellos se encargará de computar un nodo del árbol de dialéctica en forma concurrente.

Generación distribuida del árbol de dialéctica

Para realizar el proceso de generación y marcado del árbol de dialéctica en forma distribuida, un agente Ag que ha construido un argumento $\mathcal{A}$ para sustentar una conclusión h, calcula todos los puntos de ataque de $\mathcal{A}$ (denotado $PA(\mathcal{A})$). La busqueda de derrotadores para los puntos de ataque la realizan los agentes derrotadores, los cuales se distribuyen entre todos los sitios del sistema, siguiendo alguna política de distribución. Esto es, para cada punto de ataque $p_i \in PA(\mathcal{A})$, se solicita, a través de un servicio especial, la creación de un agente derrotador AD_i en un sitio determinado. La función de este agente AD_i es atacar al argumento $\mathcal{A}$ en el punto p_i que se le asigne, y luego informar al agente Ag si puede o no derrotar a $\mathcal{A}$. Una vez que se han distribuido todos los puntos de ataque a diferentes agentes derrotadores, el agente Ag debe esperar que los agentes derrotadores respondan. Pueden darse dos casos:

1. Si uno de los agentes AD_i responde afirmativamente, es decir, AD_i pudo construir un derrotador para $\mathcal{A}$, entonces:
   (a) la conclusión h no se puede garantizar
   (b) no es necesario seguir esperando la respuesta del resto de los agentes derrotadores, los cuales pueden ser eliminados para evitar cómputo innecesario (poda de un sub-árbol).
2. Si ninguno de los agentes AD_i responde afirmativamente, entonces h estará garantizado.

[1] En inglés *undefeated*

Agentes Derrotadores

El objetivo de estos agentes es intentar derrotar un argumento $\mathcal{A}$, atacándolo en un punto dado p, como respuesta al pedido de otro agente. Para hacer esto un agente derrotador AD_i debe encontrar un derrotador $\mathcal{D}$ para $\mathcal{A}$, que no sea derrotado. Para saber si $\mathcal{D}$ es derrotado o no, el agente AD_i debe calcular los puntos de ataque de $\mathcal{D}$, distribuirlos entre los sitios del sistema, y esperar la respuesta. La distribución de estos puntos de ataque, crea nuevos agentes derrotadores, que pueden a su vez generar mas agentes derrotadores recursivamente. Si el derrotador $\mathcal{D}$ encontrado no tiene puntos de ataque, entonces $\mathcal{D}$ no puede ser derrotado y AD_i puede responder al agente que lo solicitó (i.e. a su padre) que derrota a $\mathcal{A}$. Si $\mathcal{D}$ tiene puntos de ataque, una vez que AD_i los distribuyó, debe esperar que sus hijos (i.e. agentes solicitados por él) respondan. Pueden darse tres casos:

1. Si uno de estos agentes solicitados por AD_i responde que derrota a $\mathcal{D}$, entonces:
   (a) el derrotador $\mathcal{D}$ debe ser descartado,
   (b) no es necesario seguir esperando la respuesta de los demás hijos que intentan derrotar a $\mathcal{D}$, y pueden ser terminados para evitar computo innecesario (poda de un sub-árbol).
   (c) AD_i debe buscar para el punto de ataque p que le fue asignado, otro derrotador $\mathcal{D}'$ para $\mathcal{A}$ y verificar con este mismo proceso que $\mathcal{D}'$ no sea derrotado generando agentes derrotadores para $\mathcal{D}'$.
2. Si todos los agentes solicitados por AD_i responden que no derrotan a $\mathcal{D}$, entonces AD_i puede responder, al agente que lo solicitó (padre), que **derrota** a $\mathcal{A}$ en el punto p.
3. Si el agente AD_i no logra encontrar un derrotador para $\mathcal{A}$ que no sea derrotado, entonces responde que **no derrota** a $\mathcal{A}$ en el punto p.

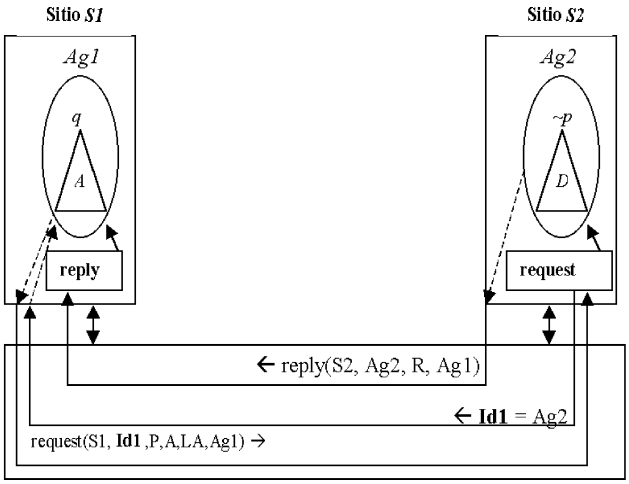
Dado que los agentes deben conocer quien los solicitó para poder responderles, entonces cada agente tendrá un identificador único dentro del sitio donde se esta ejecutando.

Implementación

Para implementar el sistema multiagentes se utilizó BinProlog [7], un interprete de Prolog con capacidad *multi-thread*. BinProlog permite ejecutar servidores, los cuales pueden atender concurrentemente consultas locales o remotas, y manejar un blackboard sincronizado. Los sitios donde se ejecutan los agentes fueron implementados como un proceso del interprete BinProlog en el cual se corre un servidor que escucha en un determinado port. A este se agrega un interprete de DeLP, y un conjunto de servicios especiales implementados en BinProlog para la comunicación y sincronización de los agentes. La dirección de un sitio está formada por la dirección de red del host donde se esta ejecutando mas el número de port donde escucha el servidor de BinProlog.

Los sitios pueden estar en una sola computadora, siempre y cuando escuchen en diferentes ports, o pueden correrse en diferentes maquinas que estén conectadas por una red. Si un sitio S_1 necesita invocar algún servicio de otro sitio S_2, simplemente ejecuta remotamente en S_2 el predicado que implementa el servicio. La información que se necesite enviar, como parte del pedido, viaja en los argumentos de dicho predicado

Si un agente Ag_1 que se encuentra en el sitio S_1 desea solicitar la creación de un agente derrotador Ag_2 que intente derrotar a un argumento $\mathcal{A}$ en un punto de ataque P, y Ag_2 debe ejecutarse en el sitio S_2, entonces se ejecuta remotamente en S_2 el predicado `request(D1, Id1, P, A , LA, Id2)`. El parámetro `D1` es la dirección del sitio S_1, `Id1` el identificador del agente Ag_1, `P` el punto de ataque, `A` es el argumento a derrotar, `LA` la línea de argumentación, y `Id2` el identificador del nuevo agente creado en S_2 como respuesta al pedido. Al ejecutarse el predicado `request` se crea un nuevo agente derrotador Ag_2 en un nuevo thread dentro del sitio S_2. Este nuevo agente intentará construir un derrotador para el punto `P` de `A`. Cuando el agente derrotador Ag_2 finaliza su tarea en el sitio S_2, y quiere responder al agente Ag_1 que lo creó desde el sitio S_1 con dirección `D1`, entonces ejecuta remotamente en `D1` el predicado `reply(D2,Id2,R,Id1)`, donde `D2` es la dirección de el sitio S_2, `Id2` es el identificador de Ag_2, `R` es la respuesta (derrota o no derrota), y `Id1` es el identificador de Ag_1.



Como se explicó anteriomente, un argumento $\mathcal{A}$ puede tener más de un punto de ataque, y para rotular a $\mathcal{A}$ como derrotado o no derrotado, se debe esperar hasta que uno de sus agentes derrotadores responda que derrota a $\mathcal{A}$ en uno de esos puntos, o hasta que todos sus agentes derrotadores respondan que no derrotan a $\mathcal{A}$. Por lo tanto, cuando un agente Ag_1 construye un argumento $\mathcal{A}$ se genera el conjunto de puntos de ataque y se distribuyen en los sitios disponibles creando los agentes derrotadores. El agente Ag_1 almacena en su sitio la información sobre cada agente derrotador creado, y luego se suspende a la espera de las respuestas. Quien se encarga de despertar al agente cuando uno de los agentes derrotadores Ag_2 responde es el predicado `reply(D2,Id2,R,Id1)` que realiza lo siguiente:

1. Recupera la información almacenada por Ag_1 sobre Ag_2, utilizando `D2,Id2,Id1` y registra que Ag_2 ha respondido `R`.
2. Si la respuesta es afirmativa `R=yes` entonces:
   (a) despierta a Ag_1 para informale que ha sido derrotado
   (b) para evitar computo innecesario, termina con todos los demás hijos (derrotadores) que aun no han respondido mediante el servicio `shutDownAgent`, y elimina del sitio S_1 toda la información almacenada sobre los derrotadores generados para Ag_1.

3. Si la respuesta es negativa `R=no`, y ya respondieron negativamente todos los demás derrotadores entonces despierta a Ag_1 para informale que no puede ser derrotado en ninguno de sus puntos de ataque.

Observese que el paso 2b corresponde a una poda del árbol de dialéctica que se realiza en forma distribuida. El servicio `shutDownAgent(Id)` se ejecuta en el sitio donde se encuentra el agente `Id` y antes de eliminar al agente `Id`, debe eliminar a todos los derrotadores que puedan haberse generado para `Id`, y así recursivamente. Con lo cual `shutDownAgent(Id)` elimina todo el posible sub-árbol de raiz `Id` que se pueda haber generado hasta ese momento, con el objetivo de rotular al argumento de `Id` como derrotado o no derrotado.

Conclusiones

Las principales ventajas de esta arquitectura son eficiencia y robustez. Aunque la generación del árbol de dialéctica se realiza en profundidad, el paralelismo logrado por los agentes permite calcular cada rama del árbol en paralelo. Este paralelismo incorpora una componente de generación del árbol a lo ancho, que permite encontrar más rápidamente cualquier rama de pocos niveles que permita decidir si un nodo está derrotado. El sistema también resulta mas robusto, ya que si un sitio se cae, pueden reasignarse las tareas en otro sitio. La implementación realizada, ante la caída de un sitio, reasigna todos los agentes de ese sitio en otros que esten funcionando.

Referencias

[1] Grigoris Antoniou, Michael J. Maher, and David Billington. Defeasible logic versus logic programming without negation as failure. *Journal of Logic Programming*, 42:47–57, 2000.

[2] A. Bondarenko, P.M. Dung, R.A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93:63–101, 1997.

[3] Phan M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.

[4] Alejandro J. García. *Programación en Lógica Rebatible: Lenguaje, Semántica Operacional, y Paralelismo.* (`http://cs.uns.edu.ar/~ajg`). Universidad Nacional del Sur, Bahía Blanca, Argentina, 2000.

[5] John Pollock. Implementing defeasible reasoing. *workshop on Computation Dialectics*, 1996.

[6] Henry Prakken and Giovanni Sartor. Argument-based logic programming with defeasible priorities. *J. of Applied Non-classical Logics*, 7(25-75), 1997.

[7] Paul Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.