Э. А. Трахтенгерц, Взаимодействие агентов в многоагентных системах, *Автомат. и телемех.*, 1998, выпуск 8, 3–52

Автоматика и телемеханика, № 8, 1998

Обзоры

УДК 681.3

© 1998 г. **Э.А. ТРАХТЕНГЕРЦ, д-р техн. наук**
(Институт проблем управления РАН, Москва)

ВЗАИМОДЕЙСТВИЕ АГЕНТОВ В МНОГОАГЕНТНЫХ СИСТЕМАХ

В обзоре дается характеристика многоагентных систем, устанавливается связь между понятиями, используемыми в теории интеллектуальных систем, и терминологией, используемой в объектно-ориентированном программировании, рассматриваются методы обмена информацией между пользователями, языковые средства описания и функционирования агентов и особенности управления вычислительным процессом в многоагентных системах.

1. Структура многоагентных систем

Как только появились вычислительные машины начались работы по созданию "общего языка" человека и ЭВМ. Результаты этих работ хорошо известны: удалось создать дружественный интерфейс, обеспечивающий эффективное взаимодействие человека и компьютера. Эти работы интенсивно продолжаются и одним из последних результатов является технология многоагентных систем.

Можно сказать, что в этом "общем языке" заложены две составляющие:

– методы и алгоритмы, посредством которых многоагентная система решает задачи, анализирует обстановку, вырабатывает рекомендации, т.е. методы вычислений, поиска информации и генерации решений;

– методы создания агентов, алгоритмы их взаимодействия и организация функционирования системных программ многоагентных систем поддержки принятия решений.

Обзор посвящен второй составляющей.

При практической реализации распределенных систем, в частности систем поддержки принятия решений (СППР), возникли серьезные трудности с проектированием и даже просто описанием объединенных в единую сеть разнородных локальных компьютерных узлов. Эти узлы воспринимают из внешнего мира, в том числе и от человека, различную информацию, обмениваются данными друг с другом, перерабатывают эти данные в соответствии с заложенными в них алгоритмами и в результате вырабатывают некоторые рекомендации или решения. В последние годы в рамках общего научного направления "искусственный интеллект" активно ведутся исследования под объединенным названием "многоагентные системы" [1–4]. Упор на многоагентные, т.е. распределенные, системы сделан в связи с тем, что в системах искусственного интеллекта из-за огромного объема вычислений, связанного, в частности, с необходимостью осуществлять большой перебор, например, возможных ходов при игре в шахматы, приходится использовать мощные распределенные многопроцессорные вычислительные комплексы и сети. Так, вычислительная машина Deep Blue, игравшая с Г. Каспаровым, состоит из 32 высокопроизводительных процессоров.

Интерес к многоагентным системам возрастает по следующим причинам [5, 6]:

– в связи с необходимостью решать задачи настолько сложные и "большие", что они не могут быть решены одним агентом как из-за ограниченности ресурсов, так и из-за риска отказа централизованной системы;

– из-за удобства организации взаимодействия и обмена информацией между функционирующими информационными системами, системами поддержки принятия решений, экспертными системами и т.д.;

– для повышения эффективности работы естественно распределенных систем (таких как управление воздушным движением, банковские системы, автоматизированные системы управления производством), систем, связанных с пространственно распределенными источниками информации, и систем, в которых пространственно распределены профессиональные знания (например, медицинские системы);

– для сокращения времени обработки информации (за счет параллельных вычислений), повышения надежности (способности функционирования при отказе отдельных компонентов), перестраиваемости системы (способности менять число процессоров);

– для обеспечения концептуальной простоты и простоты разработки;

– в связи с тем, что стратегия развития вычислительной техники конца этого века и, видимо, начала будущего ориентирована в основном на сетевые вычислительные структуры, в которых задачи решаются распределенно;

– так как принцип модульного построения и объектно-ориентированное программирование, ставшее ведущей технологией разработки программ, хорошо согласуется с многоагентной идеологией построения интеллектуальных систем.

Для того, чтобы уяснить себе организацию компьютерного взаимодействия в многоагентных системах, рассмотрим инструментальные (языковые) и системные (управляющие вычислительным процессом) программные средства многоагентных систем. Поскольку в литературе по многоагентным системам все шире используется специальная терминология, рассмотрим эту терминологию.

Слово "агент" имеет широкий диапазон значений: от "агента влияния", которым клеймят своих противников политические деятели, до безобидных программных модулей в сложных компьютерных системах. Нас будут интересовать программные модули. Словарь Вебстера дает следующее определение агента: "лицо или фирма, облеченные полномочиями действовать за другого". В этом определении нет упоминания программного модуля в качестве агента, но модуль в многоагентной системе тоже уполномочен действовать "за другого" – эксперта или лица, принимающего решение (ЛПР). Термин "агент" является полезной метафорой для агентно-ориентированных систем, являющихся объединением объектно-ориентированной технологии программирования и технологии искусственного интеллекта [4]. Действительно, с инженерной точки зрения агентно-ориентированное программирование может рассматриваться как специальный класс объектно-ориентированного программирования. Очень важно, что это именно объектно-ориентированная технология программирования, так как она становится, или уже стала, основной технологией создания программного обеспечения. Особенность агентно-ориентированного программирования состоит в том, что фиксируются состояния модулей (теперь называемых агентами) с помощью определенных компонентов, называемых убеждениями (beliefs), возможностями, выбором и, если необходимо, другими подобными характеристиками. Вычисление состоит из обработки информации, посылки требований, предложений, выражения согласия, отказа, организации состязания, помощи друг другу и, наконец, выработки решения [7].

Многоагентные системы должны отвечать [3]:

а) современным стандартам программирования:

– модульности, обеспечивающей уменьшение сложности и облегчающей разработку, тестирование и эксплуатацию;

– эффективности, обеспечивающей быстрое выполнение и нахождение решений путем параллельной реализации процессов;

– повторному использованию для избежания избыточности и дублирования разработок;

б) следующим требованиям функционирования:

– включать в себя агенты, выполняющие различные функции;

– обеспечивать взаимодействие агентов;

– обеспечивать когерентность деятельности агентов и адекватное глобальное поведение системы.

Для лучшего понимания особенностей агента на рис. 1,*а* показана схема "классического" программного объекта (объектного модуля) [8], а на рис. 1,*б* – агента (агентного модуля, несколько измененного по сравнению с [4]). Рассмотрим сначала внутреннюю структуру объектного модуля (рис. 1,*а*). Он состоит из блока управления процессом, таблиц связи с методами, программ, реализующих методы обработки данных (на рис. 1,*а* – методы), входных и выходных очередей сообщений. Блок управления процессом содержит информацию, необходимую для управления процессом и очередями сообщений. Когда приходит новое сообщение, оно заносится в очередь сообщений, и блок управления процессом определяет методы (т.е. подпрограммы), которыми оно должно быть обработано. Затем с помощью таблиц связи с методами и указателей методов, используя выделенную ему операционной системой оперативную память, организует обработку поступившего сообщения и генерирует выходные сообщения. Как только объект переходит в пассивное состояние (т.е. как только он ответил на последнее выходное сообщение и не получил новое), выделенная ему область оперативной памяти отнимается. Штрихпунктиром на рис. 1,*а* обозначена пересылка данных, сплошной линией – команды управления.

На рис. 1,*б* показана схема агента в терминах методов искусственного интеллекта. В блоке "структура агента" появляются упоминавшиеся выше, но еще не раскрытые термины "убеждения", "желания", "замыслы и обязательства". Мы еще вернемся к этим терминам. Пока только отметим, что это аналоги методов обработки данных, показанные на рис. 1,*а*. Блок "активность агента" состоит из блока, аналогичного блоку управления процессом рис. 1,*а* – управление собственным процессом блока выполнения собственных функций – это набор методов обработки информации на рис. 1,*а* и блоков управления входными и выходными сообщениями (связь с другими агентами, связь с внешним миром). Активность "обновление своей структуры" также относится к функции блока управления процессом на рис. 1,*а*. Штрихпунктиром на рис. 1,*б* обозначена пересылка данных, сплошной линией – переход из состояния в состояние. Состояний может быть три: активное, "замороженное" (пассивное) и ликвидации. Заметим, что ликвидация агента может потребовать некоторой специальной деятельности: информации других агентов, контроля за сообщениями, посланными ликвидируемому агенту или отправленными им, пересылки базы знаний и базы данных. Переход из состояния в состояние, как правило, осуществляется операционной системой.

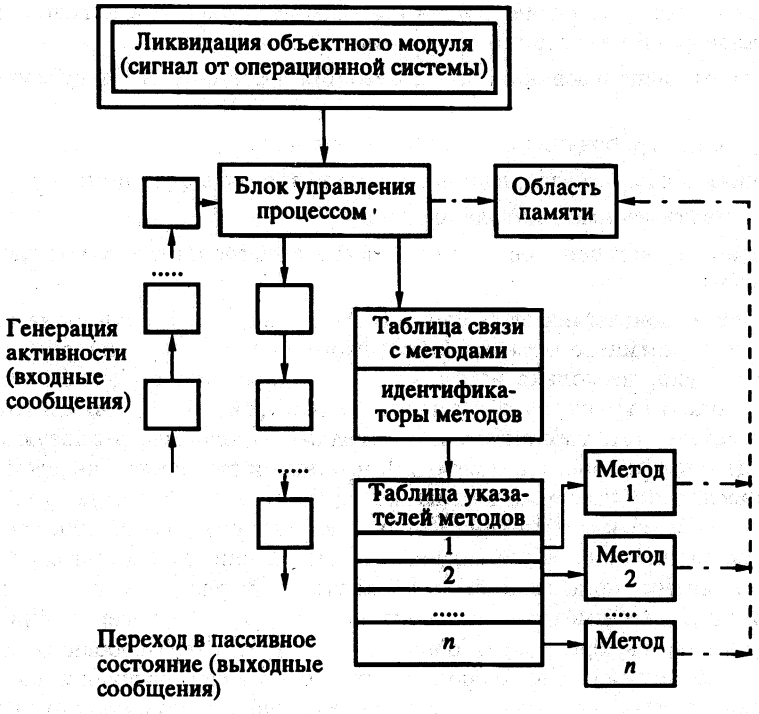
Для реализации своих функций агент должен обладать по крайней мере четырьмя возможностями [9]:

– поддерживать взаимодействие с окружающим миром, получая от него информацию и реагируя на нее своими действиями (реактивность);

– проявлять собственную инициативу (активность);

– посылать и получать сообщения от других агентов и вступать с ними во взаимодействие (социальная способность);

– действовать без вмешательства извне, в том числе и без вмешательства человека (автономность).


Ликвидация объектного модуля (сигнал от операционной системы)
Блок управления процессом
Область памяти
Генерация активности (входные сообщения)
Таблица связи с методами
идентифика-торы методов
Таблица указа-телей методов
1
2
n
Метод 1
Метод 2
Метод n
Переход в пассивное состояние (выходные сообщения)

а


Генерация активности
Активность агента
- управление собственным процессом
- выполнение своих специ-фических функций
- связь с другими агентами, использование знаний о нем
- связь с внешним миром, использование знаний о нем
- обновление своей структуры
Структура агента
- управление базой данных;
- база знаний;
- убеждения;
- желания;
- замыслы;
- обязательства.
"Замораживание" агента (переход в пассивное состояние)
Ликвидация агента

б

Рис. 1

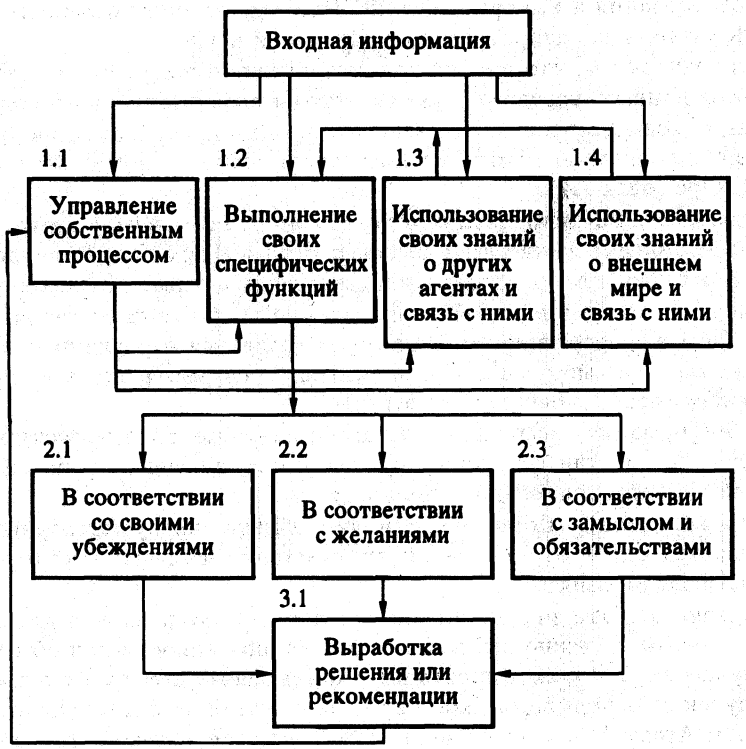


Рис. 2

С помощью рис. 2, показывающего схему функционирования агента, попытаемся отобразить термины, используемые в искусственном интеллекте, показанные на рис. 1,*б*, в понятия системного программирования. Блоками на рис. 2 показаны функции и "причины" действий агента [10], стрелками показана связь между блоками. Расшифруем составляющие этой схемы.

1.1. Управление собственным процессом. В работах по распределенным экспертным системам [6, 10] отмечалось, что экспертные системы в распределенных системах поддержки принятия решений обладают своей собственной локальной операционной системой, причем набор критериев диспетчеризации задач шире традиционного. В терминах рис. 1,*а* – это блок управления процессом или локальная операционная система, которая управляет ходом вычислительного процесса.

Естественно, локальная операционная система, управляя вычислительным процессом, определяет какие специфические функции агента должны быть выполнены в данный момент (блок 1.2 на рис. 2), т.е. какая задача должна решаться. В соответствии с рис. 1,*а* – это определение методов. Определяя используемые в задаче методы, локальная операционная система фактически определяет знания о других агентах (блок 1.3) и знания о внешнем мире (блок 1.4), которые должны использоваться в этой задаче, а также осуществляет управление процессами ввода/вывода (блоки 1.3 и 1.4).

1.2. Выполнение своих специфических функций. Ради выполнения этих функций, собственно, и создаются многоагентные системы. Каждая такая функция является составляющей в выработке решения. Так, в системах поддержки принятия решений это может быть генерация возможных вариантов решений, их оценка и/или согласование. Эти функции могут быть реализованы различными алгоритмами ге-

нерации, согласования и выбора решений. При реализации численных методов это могут быть различные подпрограммы вычислений и т.д.

Еще раз подчеркнем, что именно для выполнения этих функций создаются базы знаний и базы данных, разрабатываются методы информационного поиска, численные методы, методы распознавания, генерации, оценки, согласования решений и реализующие их программы. Этот блок использует знания о других агентах (блок 1.3) и внешнем мире (блок 1.4).

1.3. Использование своих знаний о других агентах и связь с ними. Эти знания необходимы для оценки важности и достоверности информации, получаемой от других агентов, а также при согласовании коллективных решений нескольких агентов.

Связь между агентами в любой распределенной системе представляет серьезную проблему. Обмен информацией может осуществляться в различных режимах. В качестве примера рассмотрим режимы обмена, предусмотренные в системе KAoS (Knowledgeable Agent – oriented System) [11].

Inform (информация). Это простейшая форма обмена. Агент-отправитель посылает сообщение, не требуя или требуя его подтверждения. В последнем случае агент-адресат должен подтвердить получение.

Offer (предложение). Помимо дисциплин обмена, предусмотренных в режиме Inform, возможен отказ от получения информации, если агент-адресат в момент прибытия сообщения занят.

Request (заявка). Эта дисциплина обмена лучше всего подходит для обмена сообщениями с агентом, надежно выполняющим свои обязанности при обмене. В самом простом случае агент B может просто выполнить заявку агента A с опцией подтверждения получения информации. Заявка также может быть отклонена агентом B или повторена им. Агент A, в свою очередь, может в любое время повторить заявку или отозвать ее. Если заявка агента A принимается агентом B, он посылает сообщение агенту A, что заявка принята, и сообщает результаты ее обработки.

При обмене информацией между агентами можно исходить из следующих начальных предпосылок [12]:

1. Неполная информация. Агенты могут не иметь доступа ко всем данным оппонентов, в частности, не знать их функций предпочтения.

2. Рациональность. Агенты рациональны в том смысле, что они стараются максимизировать свои функции предпочтения.

3. Выполнение соглашений. Если соглашение достигнуто, договаривающиеся стороны его выполняют.

4. Отсутствие долгосрочных обязательств. Каждое соглашение независимо, и пока оно не выполнено, не должно быть соглашений с другими агентами, изменяющих уже принятые обязательства.

5. Прекращение переговоров. Агент может прекратить переговоры, если ему это выгодно.

Алгоритмы диспетчеризации и мониторинга, рассматриваемые ниже, исходят из этих предпосылок.

Надо отметить, что за счет разницы между моментом передачи сообщения и моментом реакции на него, в любой распределенной системе возникает асинхронность в передаче сообщения. При проектировании таких систем необходимо предусмотреть, чтобы не возникали состояния, в которых обмен сообщениями между "разговаривающими" агентами может быть нарушен в результате асинхронности отправления и получения сообщений. Обмен информацией между агентами является важной функцией распределенной системы, и мы еще поговорим о ней позже.

1.4. Использование своих знаний о внешнем мире и связь с ним. Знания о мире необходимы для оценки важности и достоверности информации, получаемой из внешнего мира, а также для выбора алгоритмов вычислений. Алгоритмы вычислений могут быть определены экспертом или ЛПР, которые для системы поддержки

принятия решений являются элементами внешнего мира. Связь с внешним миром нужна для получения информации и оценки обстановки, по которой принимается решение.

Перейдем на 2-й уровень графа рис. 2. В нем отражены, если так можно выразиться, "побудительные мотивы" действий агента.

Выполнение агентом своих специфических функций с использованием знаний о других агентах и внешнем мире осуществляется в соответствии с набором заложенных в него алгоритмов, знаний и полученной информацией (теперь называемый его убеждениями (блок 2.1)). Агент определяет желаемый результат (блок 2.2) и свои намерения (блок 2.3) с помощью алгоритмов и программ, которые будут их реализовывать. Таким образом блоки 2.1, 2.2 и 2.3 являются составляющими блока 1.2.

2.1. Убеждение (belief). Словарь Вебстера дает определение термина belief – "нечто во что верят или воспринимают как истинное", т.е. этот термин не вполне эквивалентен русскому слову "убеждение". Может быть даже ближе был бы термин "вера", но в русском языке он имеет другой смысл. Поэтому будем использовать термин "убеждение" за неимением более точного эквивалента.

Агент может быть "убежден", т.е. воспринимать как истинные, правила формирования вывода в экспертных системах, базовые шкалы и "веса" критериев, функции (полезности) или отношения предпочтений, правила замещения при определении гиперповерхностей безразличия и т.д. Таким образом понятие "убеждение" применительно к агенту системы поддержки принятия решений имеет совершенно конкретный смысл.

В [10] "убеждения" подразделяются на:

– внутренние "убеждения" агента: алгоритмы и оценки, заложенные в него при разработке или внесенные в процессе эксплуатации;

– убеждения, возникшие в результате наблюдения, формируемые по правилу: ЕСЛИ наблюдается факт (X), ТО убеждение (X);

– убеждения, возникшие в результате связи с другими агентами, формируемые по правилу: ЕСЛИ A сообщает о факте (X, A) И (A) – заслуживающий доверия источник, ТО убеждение (X).

2.2. Желания по определению, данному в [11] – это цели или амбиции агента, которые он может достичь посредством совершения кажущихся ему возможных действий.

Можно сказать, что агент определяет абстрактное подпространство, в котором он находится в начальный момент своего функционирования, и определяет подпространство, в котором он "хотел бы" оказаться в результате выполненных им действий. Например, агент, регулирующий давление пара, определил, что давление превысило норму. Он предпринимает некоторые действия, скажем дает команду на открытие клапана, в результате чего давление становится нормальным. Нормальное давление в пределах соответствующих допусков – это то абстрактное пространство, в котором агент "хотел бы" оказаться.

2.3. Замыслы и обязательства (по терминологии, используемой в некоторых работах – намерения). Замыслы агента могут интерпретироваться как набор алгоритмов (сценариев), привлекательность и реализуемость которых меняются в зависимости от информации, получаемой из внешнего мира и от других агентов. Обязательства – необходимость (обязанность) перевести систему из абстрактного подпространства, в котором она находится, в абстрактное подпространство, в котором она должна находиться.

3.1. Выработка решения задачи или рекомендации осуществляется за счет оценок возможных алгоритмов (сценариев) и выбора из них наилучшего с учетом ограничений, диктуемых обстановкой и влиянием других сценариев. Результат представляется пользователю через блок 1.1.

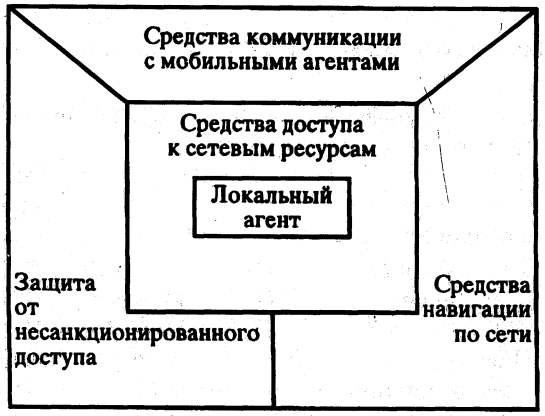


Рис. 3

Из рис. 2 видно, что выполнение агентом своих функций, показанных на первом ярусе схемы, осуществляется на основании "побудительных мотивов", показанных на втором ярусе. В схеме на рис. 2 нет функции "обновление своей структуры", показанной на рис. 1,*б*. Она осуществляется на основании информации, полученной из внешнего мира и от других агентов, и поэтому "растворилась" в блоках 1.3 и 1.4.

Необходимо отметить, что во многих работах, например [2, 5], характерной особенностью агента является его способность к обучению.

На рис. 1,*б* показана структура локального агента. Локальные агенты имеют доступ к локальным ресурсам и их иногда называют персональными помощниками. Они часто реализуют персонифицированный интерфейс, т.е. настроены на индивидуальные привычки и предпочтения конкретного пользователя [2], множество функций, которые они способны выполнять достаточно широко. Первые идеи создания таких агентов относятся еще к 70-м годам [13]. Новые реализации описаны в работе [14].

Сетевые агенты [2]. В отличие от локальных агентов сетевые агенты имеют доступ к сетевым ресурсам. Они должны быть снабжены необходимой информацией об адресах и ресурсах сети.

Мобильные агенты. Самое, наверно, грубое, но хорошо иллюстрирующее определение [15]: мобильный агент – это программа, которая может мигрировать от машины к машине в гетерогенной сети. Он может прекратить выполнение программы в произвольной точке, передислоцироваться на другую машину и продолжить выполнение программы.

В [5] дано более развернутое определение. Мобильный агент – это программный объект, существующий в программной среде. Он обладает всеми характеристиками локального и сетевого агентов, кроме того, он включает в себя средства коммуникации с другими мобильными агентами, средства защиты от несанкционированного доступа и средства навигации (перемещения) по сети. Структура мобильного агента показана на рис. 3.

Мобильные агенты функционируют в многоагентных системах, построенных с использованием методов распределенного искусственного интеллекта [16]. Они характеризуются скоординированным интеллектуальным поведением, обеспечивающим достижение общей цели, стоящей перед многоагентной системой.

Во всех распределенных компьютерных системах серьезной системной проблемой является организация параллельных вычислений, суть которых заключается в параллельном выполнении нескольких взаимодействующих программ (каждая такая программа называется процессом или потоком) на различных вычислительных

устройствах. В процессе выполнения программы обмениваются информацией и синхронизируют свои действия, т.е. ожидают, если это необходимо, готовности друг друга к выполнению совместных вычислений.

Организация параллельных вычислений – чрезвычайно сложная проблема [17–20]. Мы остановимся на следующих вопросах компьютерного взаимодействия:

– организация взаимодействия пользователей;

– языковые средства описания агентов и асинхронного управления ими в многоагентных системах;

– организация обмена сообщениями между агентами;

– диспетчеризация выполнения программ в каждом агенте и мониторинг задач в многоагентной системе.

Сначала рассмотрим структуры распределенных вычислительных систем.

2. Архитектуры многоагентных вычислительных систем

Идея распараллеливания вычислительных процессов, дающая возможность повысить производительность вычислительных систем и/или организовать обмен информацией между ее источниками и потребителями, привела к появлению многопроцессорных вычислительных комплексов различной архитектуры и сетей вычислительных машин, получивших название распределенных систем, ресурсы которых распределены в пространстве.

Асинхронные параллельные вычислительные процессы могут быть реализованы на трех типах вычислительных систем, получивших широкое распространение.

К первому типу принадлежат мультипроцессорные системы, имеющие общую память над несколькими процессорами. Структура этих систем хорошо известна. Это системы типа BURROUGHS, ЭЛЬБРУС, GRAY и др. [18, 19].

Ко второй группе относятся комплексы с раздельной памятью. Эта группа процессоров, каждый из которых имеет собственную оперативную память. Процессоры связаны между собой одним из видов быстродействующих шин. К таким системам относятся, например, вычислительный комплекс фирмы IBM – ICAP/3090 [21].

Это очень мощная система, объединяющая в различных сочетаниях мультипроцессорные комплексы IBM 3090 с векторными процессорами в единую систему. Связь осуществляется с помощью быстрой шины, по которой передаются сообщения, и адаптеров канал – канал, по которым происходит обмен данными. Эти комплексы имеют помимо локальных памятей в IBM – ICAP/3090 еще большую общую память в несколько сот мегабайт, прямой доступ к которой осуществляется либо по специальным каналам (в ICAP/3090 модель 300), либо по общей шине (в ICAP/3090 модель 400). Кроме того, каждый процессор имеет доступ ко всей дисковой памяти.

В последнее время все более широкое распространение получили транспьютеры, из которых создаются вычислительные комплексы второго типа. Одна из главных причин, обуславливающих эффективность транспьютера с точки зрения производительности и простоты реализации – это рациональный выбор его набора команд и заложенная в нем возможность создавать многопроцессорные системы [22].

Транспьютеры часто характеризуют как процессоры с сокращенным набором команд (RISC – reduced instruction-set computing). В них действительно используются преимущества RISC-процессоров. Однако они обычно содержат и небольшой набор не RISC'овых команд, т.е. команд, выполняемых не за один такт, связанных с управлением и передачей сообщений. Транспьютер может работать с собственной оперативной памятью. Обмен данными между параллельно работающими транспьютерами осуществляется посредством пересылок сообщений, процедура обмена сообщениями поддержана аппаратно.

Любой транспьютер может быть использован, как однокристальная система, содержащая процессор и оперативную память. В таком виде они часто используются во встроенных системах автоматического управления.

Для объединения двух транспьютеров достаточно их соединения шиной, так как в кристалле транспьютера встроены специальные средства обмена сообщениями, называемые линками. Существующие транспьютеры имеют по 4 канала связи (линка). Линки являются важным механизмом построения систем с множественными потоками команд и данных. Линки на различных типах транспьютеров совместимы, таким образом эти транспьютеры могут быть объединены в одну общую систему. Скорость линков может быть выбрана независимо от скорости процессора и линки одного кристалла могут работать с разной скоростью. Стандартной конфигурации при объединении транспьютеров не предложено, они могут иметь достаточно разнообразные топологии связей. Производительность транспьютеров достаточно высока.

Для построения систем с максимальным параллелизмом широко используется топология соединений, получившая название "гиперкуб". Узел гиперкуба содержит процессор, оперативную память процессора и средства связи. Каждый узел гиперкуба порядка n соединяется с $n-1$ другими узлами. Все памяти являются локальными, а слабосвязанные узлы взаимодействуют, передавая сообщения через промежуточные. Поскольку сообщения могут проходить через много узлов, для систем с архитектурой гиперкуб серьезной проблемой является задержка передачи данных.

Существует много задач, важных для приложений, которые содержат ряд алгоритмов, допускающих параллельную обработку. В таких случаях можно распараллеливать алгоритм на несколько потоков команд (несколько процессов), разбивая программу на части, реализующие различные или одни и те же алгоритмы в виде параллельных процессов, выполняемых на различных процессорах. Эти процессы могут обмениваться данными или работать с общими данными, но процессы должны быть достаточно независимыми, чтобы параллельная обработка проводилась без существенных издержек, связанных с синхронизацией, и каждый процесс должен выполняться достаточно долго, чтобы накладные расходы, связанные с инициацией и завершением процесса, составляли незначительную долю общего времени выполнения процесса. Для решения таких задач чрезвычайно эффективными оказываются только что рассмотренные вычислительные комплексы.

Если описывать параллельно выполняемые программы в терминах искусственного интеллекта, то группы параллельных процессов (или их части) можно рассматривать как агентов, обменивающихся сообщениями друг с другом (рис. 4).

Отметим, что многопроцессорные системы первых двух типов эффективны при решении задач поддержки принятия решений (или принятия решения) для одного ЛПР. Может быть, самым ярким примером такого использования сегодня является шахматный компьютер Deep Blue. Однако при подключении терминалов такие системы могут обслуживать и несколько ЛПР или экспертов.

Наконец, третий тип многопроцессорных вычислительных систем – это вычислительные сети, получившие в последние годы широчайшее распространение и ставшие такими же привычными как, например, телефон, почта, телеграф и, более того, во многом уже выполняющие функции этих средств связи.

Вычислительная сеть – это множество компьютеров, соединенных каналами связи, и снабженное соответствующим программным обеспечением. Когда мы говорим сегодня о вычислительных сетях и хотим говорить как можно короче, наверно надо говорить об Internet. На сегодняшний день она представляет собой глобальную информационную сеть, обеспечивающую ее пользователям быстрый доступ к океану различной информации. Важнейшей характеристикой сети являются виды сервиса, которые она может предложить пользователю. В настоящее время самым популярным видом сервиса в Internet является World Wide Web (сокращенно WWW). Это

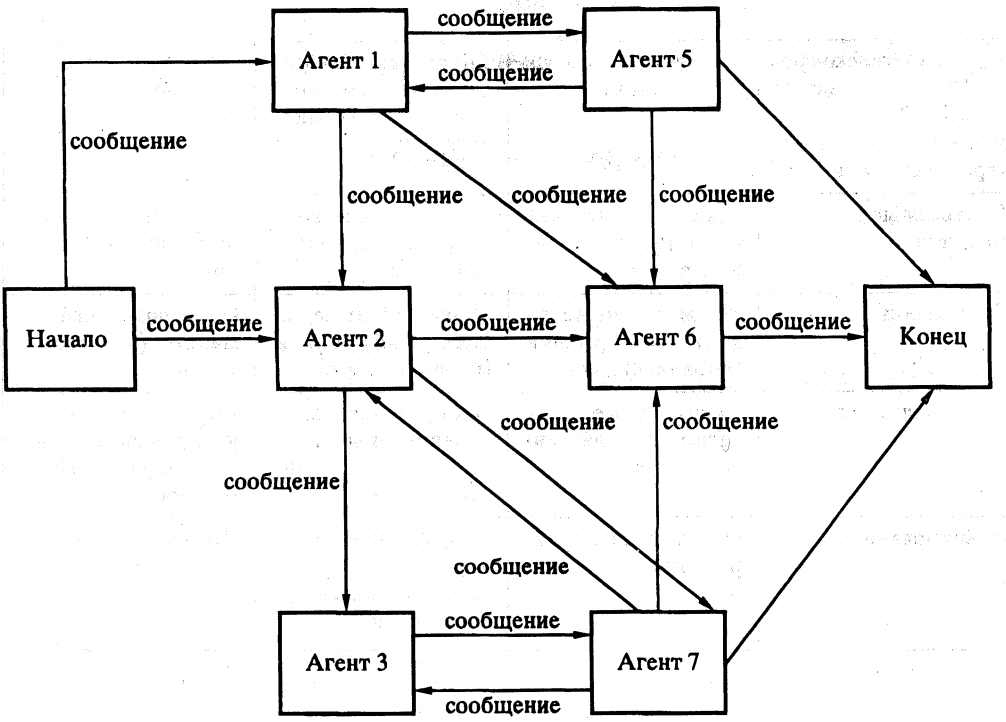


Рис. 4

английское словосочетание можно перевести как "Всемирная паутина". В течение нескольких лет эта система бурно развивалась. Так, в июле 1995 г. число серверов WWW составило 23500, из них 31,3% коммерческих, в январе 1996 г. число серверов WWW достигло 90000, из них 50,2% коммерческих, а в июне 1996 г. число серверов увеличилось до 230000 [23]. Интенсивное развитие не прекратилось и в дальнейшем.

Важно отметить, что вычислительные сети могут быть использованы и уже широко используются для принятия решений в иерархических системах управления, при принятии коллективных решений (при самой различной структуре коллектива, принимающего решения), в тех случаях, когда источники информации и/или ЛПР территориально разобщены. В этом случае каждый ЛПР или член коллектива, принимающего решения, будет располагаться на узле сети, используя ее средства для обмена информацией друг с другом.

3. Организация обмена информацией между пользователями в многоагентных вычислительных системах

Для реализации параллельных вычислений в распределенных системах потребовалось не только значительное расширение возможностей программного обеспечения, но и новое понимание вычислительного процесса. Он стал рассматриваться не как множество последовательно выполняемых действий, а как совокупность асинхронно выполняемых параллельных вычислительных процессов (в терминологии раздела 1 – параллельно реализуемых действий агентов). Такой подход позволил перейти к получающей все большее распространение групповой обработке данных, в которой при появлении новых данных или результатов у одного пользователя (в

Таблица 1

Тип телеконференции / Сравниваемые характеристики	Электронная почта	Групповая обработка данных (многопользовательский интерфейс)	Комната видеоконференций
Аппаратные средства	Персональные компьютеры, включенные в сеть	Персональные компьютеры, включенные в сеть	Аппаратура комнат видеоконференций
Параллелизм	Множественные потоки информации (параллельные)	Множественные потоки информации (параллельные)	Один поток информации (последовательный)
Синхронизация	Асинхронные процессы обмена информацией	Частично синхронизированные процессы обмена информацией	Полностью синхронизированные потоки обмена информацией
Планирование	Отсутствие планирования	Оперативное планирование в процессе обмена информацией	Предварительное планирование

одном процессе, агенте) ему предоставляется возможность немедленной рассылки этой информации другим заинтересованным пользователям (процессам, агентам). Эти возможности особенно ценны в распределенных системах поддержки принятия решений. Одна из определяющих характеристик распределенного решения – это совместное использование группы распределенных в пространстве, но связанных между собой источников знаний, когда ни один из них не имеет достаточно информации и средств для решения всей задачи. Прийти к общему решению они могут только совместно.

Быстрое развитие сети телекоммуникаций позволило использовать их для организации различного рода совместной деятельности, в том числе и для принятия коллективных решений, получило название телеконференций. Они характеризуются следующими особенностями [24]:

– телеконференции легко создаются с помощью сетей, члены телеконференций удалены друг от друга;

– в одном интервале времени один и тот же человек может входить в несколько телеконференций, в результате создаются параллельные потоки информации от различных телеконференций к одному пользователю;

– в телеконференцию могут быть вовлечены люди, не знающие друг друга.

Можно говорить о трех типах организации телеконференций: с использованием электронной почты, с использованием средств групповой обработки данных и с использованием комнат видеоконференций. Их характеристики показаны в табл. 1 [24].

Телеконференция с использованием электронной почты заключается в посылке и получении ее участниками серии сообщений по электронной почте. Активность участников не синхронизуется и не планируется. Каждый пользователь может одновременно участвовать в нескольких конференциях.

Работа в комнате видеоконференций позволяет собирать группы участников в удаленных друг от друга помещениях, снабженных оборудованием, необходимым для видеоконференции. Видеоконференция освобождает ее участников от длитель-

ных поездок для встреч на традиционных конференциях. При использовании комнат видеоконференций создается один информационный поток между всеми комнатами, активность участников полностью синхронизуется, планирование работы осуществляется обычно заранее.

Теперь остановимся подробнее на конференциях с групповой обработкой данных. Для принятия групповых решений необходимо синхронное взаимодействие, при котором происходит электронная беседа – обмен информацией в реальном времени через дисплеи. Такой обмен может быть осуществлен с помощью специального системного программного обеспечения, получившего название многопользовательского интерфейса [25]. Он реализуется на вычислительной сети.

Многопользовательский интерфейс предоставляет все необходимые процедуры для эффективного обмена информацией и принятия групповых (коллективных) решений, обеспечивая каждого участника группы возможностью использования персональной ЭВМ, подключенной к сети, а в некоторых случаях – общим большим экраном.

Многопользовательский интерфейс позволяет преодолевать трудности общения, связанные с одновременным участием пространственно распределенной группы сотрудников в научном исследовании, принятии решения, написании и совместном редактировании текста и т.д. Он дает возможность всем участникам группы вносить изменения в редактируемый текст, делать предположения по принятию решения, коллективно формулировать новые идеи. При этом информация, не нужная другим участникам группы, не появляется на их экранах и остается достоянием только "хозяина" этой информации.

Информация, представляющая интерес для нескольких участников группового принятия решений, располагается в "общих" окнах, т.е. на окне дисплея каждого участника группы, таким образом к ней имеют доступ все участники группы. "Общие" окна создаются по принципу WYSWIS (what you see is what I see – что видишь ты, то вижу я).

Необходимо сказать еще о двух дополнительных возможностях, которые в настоящее время предоставляются современными аппаратными и программными средствами. Это мультимедиа и гиперссылки [23].

Они являются не только средствами графики, но и "оживления", или в терминах Web, – анимации и озвучивания изображения. Оно становится динамичным и на экране дисплея возникает, например, картина работающего механизма, сопровождаемая соответствующим звучанием (скрежетом или усиленным шумом при перегрузках). Следует добавить, что в современных языках программирования [23] есть языковые средства, обеспечивающие программирование динамики изображения и возникновения соответствующего звучания.

Теперь о гиперссылках. Основной идеей, которая была использована при разработке системы WWW, является идея доступа к информации при помощи гипертекстовых ссылок. Система гиперссылок в WWW основана на том, что некоторые выделенные участки одного документа – это могут быть части текста или иллюстрации – играют роль ссылок на другие, связанные с ними, документы (или иллюстрации). Документы, на которые делаются ссылки, могут находиться как на локальном, так и на удаленном компьютере. При просмотре документа пользователем, гиперссылки обычно выделяются цветом и/или подчеркиванием. Переход на ссылку осуществляется с помощью мыши или клавиатуры.

Документы, на которые делаются ссылки, могут, в свою очередь, содержать ссылки на другие документы и т.д. Это чрезвычайно важный тип сервиса для системы поддержки принятия решений, так как позволяет ЛПР или эксперту быстро находить данные, необходимые ему для принятия или согласования решения.

В процессе электронной беседы пользователи должны строго придерживаться дисциплины использования общих окон и общего контекста. Нельзя всем участ-

никам одновременно вводить или изменять информацию в общих окнах, так как их действия могут быть противоречивыми. Например, один пользователь захочет сдвинуть телепойнтер вниз, а другой – вверх. Поэтому должна соблюдаться очередность ввода информации в общих окнах, аналогично тому, как это делается при селекторных совещаниях или радиообмене.

Дисциплина переговоров может быть определена заранее соглашением участников заседания и не поддерживаться системными средствами.

Однако в "эталонной модели взаимодействия открытых систем", принятой международной организацией по стандартизации в качестве стандарта для протоколов вычислительных сетей [17, 26], в протоколах 5-го (сеансового) уровня определены полномочия, которые могут быть предоставлены участнику сеанса. Таким полномочием может быть занесение данных в общее окно.

Полномочие является атрибутом сеансового соединения, который динамически присваивается одному из абонентов сеансового соединения, давая только ему право инициировать определенные услуги. Оно может быть доступным или недоступным для присвоения.

Другим способом организации дисциплины общения может быть задание приоритетов пользователей. В локальных сетях для систем реального времени (сети с многопользовательским интерфейсом можно отнести к этому классу сетей) обычно применяются детерминированные и приоритетные методы доступа к общей среде передачи данных (общему каналу) [27]. При этом стремятся удовлетворить двум требованиям:

– обеспечить ограниченное гарантированное время ожидания доступа к общему каналу для всех сообщений систем реального времени;

– уменьшить время ожидания для высокоприоритетных сообщений.

Приоритетный доступ основывается на том, что конкуренция источников за доступ к каналу является постоянным, а не случайным фактором. Поэтому данные можно передать в канал только в заведомо бесконфликтной ситуации. При приоритетном доступе источники сначала передают в канал управляющую информацию, задающую приоритеты, используя ее для получения сведений о взаимных приоритетах и для разрешения конфликтов на основе этих сведений. В результате канал занимает единственный источник с наибольшим приоритетом, который передает данные в бесконфликтной ситуации.

В каналах со случайным доступом при организации приоритетной дисциплины передачи данных источники занимают канал для передачи данных после выполнения детерминированной процедуры, гарантирующей разрешение конфликта. Эта процедура (борьба за канал) сводится к передаче в канал управляющей информации, задающей приоритеты источников, и к блокировке источников с младшими приоритетами.

В сетях с последовательным каналом может осуществляться локально-приоритетный доступ. Каждый источник передает данные в канал, ориентируясь на состояние канала в интерфейсном блоке и приоритет проходящего через него кадра данных. Источник передает кадр данных в канал при отсутствии в нем передачи или заменяя проходящий кадр на свой, если приоритет последнего выше. Это позволяет управлять временем доступа. В последовательном канале приемник имеет возможность уничтожать адресованные ему пакеты. Это позволяет передавать в канал данные одновременно нескольким источникам.

Таким образом в локальных вычислительных сетях в каналах со случайным доступом и с последовательным доступом предусмотрены аппаратные возможности для организации приоритетного упорядочения передачи сообщений. При использовании многопользовательского интерфейса желательно, чтобы "собеседники" были немногословны, но имели возможность высказаться тогда, когда хотят. Для реализации этих возможностей приоритет должен быть выше у того, чье сообщение меньше по

Таблица 2

Длина сообщения n байт	IBM PS/2-80	IBM PS/2-70	IBM PS/2-30	Olivetti-90
100	0,33	0,49	1,11	0,89
300	0,40	0,57	1,27	1,09
700	0,84	0,88	1,61	1,50
1100	1,31	1,38	1,94	2,06
1500	1,78	1,88	2,55	2,80

Таблица 3

n байт	2 станции	3 станции	4 станции
100	0,37	0,43	0,58
300	0,48	0,57	0,71
700	0,99	1,14	1,36
1100	1,51	1,73	2,00
1500	2,03	2,21	2,68

объему и перерыв между репликами больше. Таким образом, значение приоритета может быть функцией этих двух переменных.

Надо отметить, что быстродействие существующих ЭВМ и сетей вполне допускает передачу информации в режиме "электронной беседы". Это хорошо иллюстрирует табл. 2 [28], где приведено время (в миллисекундах), необходимое для посылки одного сообщения длиной в n байт для четырех различных рабочих станций на пустом сегменте сети Ethernet. Хотя в табл. 2 приведены данные об уже устаревших моделях ЭВМ, но суть дела они проясняют. В адаптере в процессе эксперимента использовался только один буфер, протокол 802 и метод доступа CSMA/CD. Из табл. 2 видно, что выбор рабочих станций может существенно повлиять на скорость пересылки сообщений.

В табл. 3 [28] показано время пересылки одного кадра длиной в n байтов при одновременной работе 2, 3 и 4 станций IBM PS/2-80 на сети Ethernet. Время дано в миллисекундах.

Из табл. 3 видно, что с ростом длины кадра и числа станций, естественно, увеличивается время пересылки, но в приведенном диапазоне, даже для не очень высокопроизводительной ЭВМ, оно не выходит за допустимые пределы.

В сетях с кольцевой топологией время задержки передачи кадра зависит от длины сети, числа станций, интенсивности использования сети и некоторых других параметров. С их помощью можно спроектировать локальную вычислительную сеть с нужными характеристиками.

Эксперт или ЛПР, пользуясь многопользовательским интерфейсом, не должен знать структуру и протоколы работы сети, так же как пользователь персональной ЭВМ не должен знать ее устройства, хотя основные принципы работы ЭВМ, да и сети, желательно знать.

На рис. 5 показана принципиальная схема сети, как ее может представить пользователь. Все узлы сети, с которыми работают ЛПР (они обозначены $W_0, W_1, \ldots, W_E$), в представлении пользователя подключены к некоторой системе, которая называется "сеть". На рис. 5 она обозначена пунктирной линией. Для пользователя неважно сколько и какие аппаратные средства используются для серверов, входящих в эту

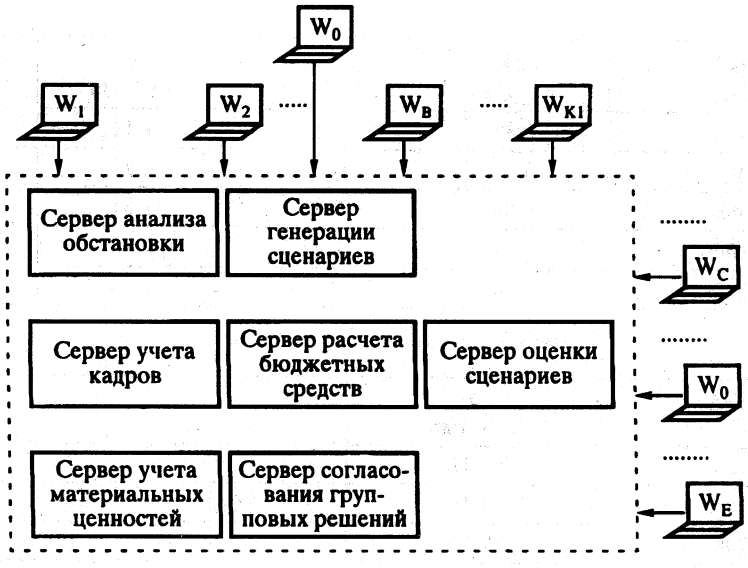


Рис. 5

сеть, какова протяженность линий, сколько подсетей входят в сеть и аналогичные подробности. Но для него важно, к каким узлам, в которых находятся ЛПР или эксперты, он может обратиться. Их имена он должен знать. Для него важно какие задачи он может решать на сети. Список задач на рис. 5 определен названием серверов. Конечно, пользователь должен знать язык общения с многопользовательским интерфейсом. Обычно это язык меню, управляемый мышью и/или клавиатурой. Меню во многих случаях может дополняться пользователем.

4. Языковые средства описания агентов и управления ими в многоагентных системах

В связи с необходимостью отразить в языках программирования новые аппаратные и программные возможности вычислительных машин сформировались три подхода к созданию языков программирования для многопроцессорных вычислительных машин и сетей [19]:

– расширение существующих широко распространенных языков программирования;

– создание новых языков программирования для конкретных типов вычислительных машин, конструкции которых могли бы эффективно транслироваться в систему команд данных машин;

– создание новых языков программирования, не ориентированных на какую-либо конкретную вычислительную систему, но содержащих новые концепции параллельного программирования.

Достоинством первого подхода является широкое распространение языка программирования, знакомство с ним пользователей и возможность использования ранее разработанных программ; недостатком – необходимость сохранения концепции используемого языка программирования, мешающей в некоторых случаях созданию языковых средств параллельного программирования.

Достоинством второго подхода является возможность создания достаточно эффективных языковых средств параллельного программирования, а недостатками –

узкая специализация на конкретный тип машин, неэффективность переноса программ, написанных на этом языке, на другие машины и необходимость изучения этого языка широким пользователем. Такие языки редко получают широкое распространение.

Достоинством третьего подхода является возможность введения новых концепций, повышающих эффективность языка программирования и надежность написанных на нем программ. Недостатком является непроверенность новых концепций, зачастую сложность создания эффективных объектных программ при трансляции с такого языка и, конечно, необходимость изучения широким пользователем не только синтаксиса языка, но и заложенных в него концепций.

Но от какого бы подхода не отталкивались разработчики можно выделить языковые средства параллельного программирования, типичные для определенных структур распределенных систем [29].

Для вычислительных многопроцессорных систем с общей памятью (см. раздел 2) в языки программирования должны быть встроены:

1. Средства описания параллельных процессов.
2. Средства генерации параллельных процессов и их ликвидации.
3. Средства синхронизации параллельных процессов.
4. Средства синхронизации доступа к разделяемым ресурсам.

Для вычислительных систем с раздельной памятью и сетей (см. раздел 2) языки программирования должны содержать следующие средства:

1. Описания параллельных процессов.
2. Генерации, ликвидации и синхронизации удаленных параллельных процессов.
3. Обмена сообщениями и их синхронизации.
4. Сохранения целостности распределенных баз данных.

Таким образом, принципиальная разница в средствах управления параллельными процессами систем с общей и раздельной памятью заключается в том, что в системах с раздельной памятью управление параллельными процессами осуществляется с помощью сообщений, а в системах с общей памятью – записью соответствующей информации в общую память.

Теперь перейдем к рассмотрению типичных конструкций языков параллельного программирования.

А. Языковые средства описания агентов (параллельных процессов). В разделе 1 было показано, что агент может рассматриваться как программный модуль, который способен обновлять структуру данных и даже знаний, посредством которых он выполняет свои функции и проявляет свою активность. В распределенных мультиагентных системах агенты могут рассматриваться как процессы или как составляющие процессов.

Агент (процесс или поток) – это последовательность операций при выполнении программы (или ее части) и данные, используемые этими операциями. Он является единицей диспетчеризации и потребления ресурсов системы.

Каждый последовательный процесс, выполнение которого возможно параллельно с другим процессом, может быть описан как задача, процедура, агент, объект, модуль, подпрограмма, функция или блок, имеющие некоторый специальный признак, который указывает, что данная часть программы может выполняться одновременно с некоторыми другими частями этой программы. В процедурных языках параллельного программирования таким признаком может быть служебное слово, например PAR, PARBEGIN, какие-либо символы, дополняющие уже используемое служебное слово (например, PSUBROUTINE, COBEGIN, COEND), дополнительные ключевые параметры, наличие которых в списке параметров показывает, что эта часть программы может выполняться параллельно с некоторыми другими частями этой программы. Такими параметрами могут быть, например, время выполнения процесса и его приоритет. Возможно введение и других признаков [19].

Для многопроцессорных вычислительных систем с общей и раздельной памятью такое описание параллелизма было вполне достаточным. С появлением вычислительных сетей и объектно-ориентированного программирования перечисленных выше простых средств стало уже недостаточно. Возникли новые требования. Ярким выразителем этих новых требований стал язык Java [30–32], в котором они нашли свое, кажется наиболее полное отражение.

В [30] дано такое определение языка программирования Java. "Java – это простой, объектно-ориентированный, распределенный, интерпретирующий, живучий, безопасный, архитектурно-нейтральный, переносимый, высокопроизводительный, многопоточный и динамичный язык. Интернет является для него и родным домом." Почти все эти эпитеты, характеризующие Jav'у, справедливы. При описании процессов (в языке Java они называются потоками – Threads) в объектно-ориентированных языках процесс может быть описан как объект, в интересующих нас системах искусственного интеллекта как агент, поскольку агент – один из видов объекта. В терминах языка Java объект – это класс. Класс – это данные и методы их обработки. Для агента такие данные и методы показаны на рис. 1,*б*.

Доступ к подобным наборам данных осуществляется через множество интерфейсов сети. Интерфейсы определяют, кто и какой тип доступа получает к данным и какие операции могут быть выполнены над ними.

Таким образом, класс – это набор элементов данных, которые включают в себя свои собственные методы (подпрограммы) их обработки. Вместо передачи данных для обработки внешней процедуре посылается сообщение для элемента данных класса, приказывая ему вызвать один из его методов, обрабатывающих данные. Заметим в скобках, что это возможно самая трудная для восприятия идея для тех, кто имеет опыт процедурного программирования, поскольку она требует полной перемены представления о том, как разрабатывать программы. Но это похоже на реальный мир: объект (класс, агент) имеет свойства (элементы данных) и поведение (методы их обработки, которые реагируют на события).

Когда выполняется программа, написанная на языке Java, расположенная на чьем-либо участке Web, то участок, где расположена программа (узловой компьютер), посылает одно или несколько описаний классов на компьютер-клиент, реализующий программу. Описание класса (в нашем представлении агента) это единственный элемент, который может путешествовать между узловым компьютером и интерпретатором Java на компьютере-клиенте. Таким образом, работа интерпретатора весьма упрощается, поскольку единственный тип сообщения, который должен "знать" интерпретатор Jav'ы для приема класса, имеет форму: "Я класс такой-то и такой-то. Вот мои переменные, а вот мои методы их обработки".

Таким образом, класс в языке программирования Java очень близок к понятию агента в теории искусственного интеллекта. Заметим, что Java не содержит в себе мобильных агентов. Это эффективная технология, позволяющая создавать мобильные агенты. В этом смысле Java вызывает к себе все больший интерес. В табл. 4 показана система Java, использующая технологию мобильных агентов. Технологии, использующие понятие мобильного агента, стали активно разрабатывать, начиная с 1994 г. Практически полезные системы, созданные с использованием технологии подвижных агентов, появились в самом конце 1996 г., в начале 1997 г. [5]. Большинство этих систем построено на базе систем программирования Java или Tcl/tk [33, 34]. В табл. 4 дается список наиболее известных систем, использующих технологию мобильных агентов.

Obliq [35] – язык программирования, ориентированный на поддержку распределенных объектно-ориентированных вычислений. Язык Obliq обеспечивает использование параллельных процессов в пределах одного адресного пространства, многозадачность, реализацию программ в сети на платформах с различной архитектурой, а также работу в Интернете. Объекты Obliq могут странствовать по сети, обеспечивая реализацию технологии мобильных агентов.

Таблица 4 [5]

Многоагентные системы	Языки агентов	Фирма-разработчик
Java	*	Sun Microsystem
Tcl/tk	*	Sun Microsystem
Obliq	*	Digital Equipment (только разрабатывается)
Aglet	Java	IBM Japan
Agent Tcl	Tcl/tk	Dartmouth College

*Java, Tcl/tk и Obliq это системы, которые включают в себя встроенные конструкции мобильных агентов. Их основу составляют программные системы того же названия.

Система Aglet [5] является следующим шагом в развитии выполняемого контекста в Интернете. В ней используется программный код, который может быть передан вместе с информацией о его статусе (состоянии). Аглеты являются объектами Java и могут перемещаться с одного узла Интернет на другие. Выполнение аглета на одной машине сети Интернет может быть прервано, при этом он может быть переслан на другой узел, где и будет продолжено его выполнение. Концептуально аглет это мобильный агент, поскольку он поддерживает возможности автономного выполнения и перемещения по сети в динамике реализации программ аглета.

Система Agent Tcl [36] это простая платформонезависимая система подвижных агентов. Она построена на базе языка программирования Tcl/tk. Навигационная модель построена на базе единственной команды agent_jump. Эта команда может появиться в любом месте агента и переводит его в "замороженное" состояние, перемещая в указанный в команде узел сети. Обмен информацией между агентами осуществляется посредством трех команд: agent_send, agent_recieve и agent_meet.

Поскольку язык Java получил несравненно большее распространение, чем другие языки, используемые при реализации технологии мобильных агентов, рассмотрим некоторые его конструкции.

В качестве самого простого описания класса (это пример класса, порождающего и оценивающего решения) приведем программу PreferenceRelation (отношение предпочтения):

```
class PreferenceRelation {
    public static void main ( ) {
        system.out. println ("Preference Relation is ")
    }
}
```

Обратите внимание на слово main во второй строке текста. Оно указывает точку входа в приложение. Main – это место, где начинается программа. В терминологии Java стартовая точка называется главным методом (main method) приложения. Существуют и другие методы инициации приложений.

Последняя строка программы – вывод слов "Preference Relation is " и оставлено место для значения отношения.

Заметим, что класс может содержать в себе другие классы – "подклассы", т.е. агент может содержать в себе другие "подагенты".

Термины public и static называются модификаторами. Они не обязательны, термин void показывает, что программа не возвращает никаких данных.

Каждый класс (агент) имеет свое имя (идентификатор). В нашем примере таким идентификатором является PreferenceRelation. Под этим именем он должен быть сге-

нерирован, активирован, переведен в "замороженное" состояние (см. рис. 1) и ликвидирован. По этому же имени к нему могут обращаться другие агенты, и он может посылать сообщения другим агентам (классам) со своим именем агента-отправителя. Как всякий класс агент может инициировать выполнение новых модулей-агентов.

В языке программирования Java главный поток и порожденные им потоки различаются синтаксическим описанием. В описании главного (порождающего) потока присутствует отличающее его служебное слово main в заголовке потока. Например:

```
public class BestDecisionThreads {
      public static void main {
            Scenario generation ( ). start ( );
                  DecisionEvaluation. start ( );
            }
}
class ScenarioGeneration extends Thread {
       public void ran ( ) {
                 ⋮
       system.out.println ("Scenario is ")
            }
}
class ScenarioEvaluation extends Thread {
       public void r ( ) {
                 ⋮
       system.out.println ("Evaluation of scenario is ")
            }
}
```

В этом примере описан главный поток BestDecision (лучшее решение) и два порожденных им потока ScenarioGeneration (порождение сценария) и ScenarioEvaluation (оценка сценария). Таким образом, процедура выбора решения описана двумя параллельно выполняемыми потоками: порождение сценария и его оценка. Оба эти потока порождаются главным потоком "лучшее решение" и будут выполняться под его управлением. Поскольку в языке программирования Java каждый параллельный поток это класс (агент), то можно сказать, что описанные процедуры выбора решения выполняются тремя агентами.

Заметим, что это, конечно, не программа, а только схема программы, показывающая принцип описания параллельных потоков в Jav'e.

В. Языковые средства инициации, завершения и синхронизации асинхронной параллельной работы агентов (процессов). Функция синхронизации процессов, происходящих в вычислительных системах, возникла с появлением мультипрограммирования в связи с необходимостью управления разделяемыми ресурсами. Особенно большое значение синхронизация приобрела в связи с появлением распределенных многопроцессорных систем.

Каждый агент может иметь ресурсы, которыми он владеет монопольно, и ресурсы, которые он разделяет с другими процессами. На логическом уровне такими ресурсами являются, например, данные. Задача синхронизации заключается в организации доступа параллельно и асинхронно функционирующих агентов к разделяемым ресурсам таким образом, чтобы они не искажали результаты вычислений.

Синхронизация обеспечивает заданную дисциплину доступа к требуемым ресурсам. Методам синхронизации посвящено очень большое число работ. Как правило, механизмы синхронизации основаны на концепции взаимного исключения доступа к ресурсу во времени, т.е. они исключают одновременный доступ к разделяемому ресурсу нескольких процессов. Например, в большинстве случаев нельзя допускать,

чтобы одновременно один процесс записывал информацию в массив, а другой – считывал эту информацию.

Простейшими языковыми механизмами синхронизации являются семафоры, введенные Дейкстрой [37]. Семафор – это неотрицательная переменная, на которой определены две операции P и V (P является первой буквой датского слова "passeren", означающего "пропустить", а V – первой буквой "vrygeven" – "освободить"). Если семафор может принимать только два значения: 0 или 1, то он называется двоичным.

Операция P на семафоре S выполняется следующим образом. Проверяется значение S. Если $S > 0$, то S: $S - 1$ и операция P считается завершенной. Если $S = 0$, то значение S не изменяется и операция P не завершается до тех пор, пока при помощи операции V значение S не станет больше 0.

Операция V изменяет значение семафора S: $S + 1$.

Заметим, что если в некотором процессе выполняется операция P над семафором S (обозначим ее $P(S)$) и значение $S = 0$, то эта операция может завершиться только, если некоторый другой процесс (обязательно другой!) при помощи операции V над тем же семафором S (обозначим ее $V(S)$) изменит значение S и сделает его больше 0.

Очень важно, что операции P и V являются неделимыми. Это означает, что выполнение операций P и V не может быть прервано до их окончания, а доступ к семафору S может быть осуществлен только посредством этих операций.

Механизм семафоров универсален и обеспечивает взаимодействие любых процессов. Распространенным механизмом синхронизации при помощи семафоров является аппарат "событий". Переменные типа "событие" являются аналогом переменной типа семафор. Для работы с переменными типа события вводятся операторы "объявить событие" и "ждать событие". Оператор "объявить событие" является аналогом операции V(X). Он часто имеет вид POST X или SIGNAL X, где X – переменная типа события.

Оператор "ждать событие" является аналогом оператора P(X) и обычно имеет вид WAIT X.

Оператор типа POST X отмечает, что событие произошло, и по заданной дисциплине (они могут быть очень разнообразны) позволяет выполняться задержанным процессам (агентам).

Оператор типа WAIT X в зависимости от того, произошло или нет ожидаемое событие, задерживает выполнение процесса (агента).

Идеология семафоров и соответствующий им синтаксис удобен для системных программ, реализующих механизм синхронизации, но не удобен в языках программирования высокого уровня. В них используются другие синтаксические конструкции.

Операторы типа RETURN, JOIN и COEND (ВОЗВРАТ, ОБЪЕДИНЕНИЕ, КОНЕЦ ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ) являются фактически синхронизирующими примитивами типа семафора. При их использовании процесс, порождающий асинхронные параллельные процессы, проверяет при помощи семафора, закончилось ли выполнение порожденных им процессов, и если нет, то ожидает их завершения, а если они закончились, то продолжается выполнение порождающего процесса.

Операторы инициации параллельных процессов также являются операторами синхронизации.

Языковые средства инициации процесса часто бывают синтаксически близки к языковым средствам обращения к подпрограммам или процедурам в данном языке. В фортраноподобных языках, например, для инициации процессов используется служебное слово CALL, обычно с некоторыми дополнительными символами, например, PARCALL, далее следует имя процесса и параметры. Асинхронный параллельный процесс рассматривается как подпрограмма или функция, но выполняющаяся параллельно с вызывающей программой.

Однако, несмотря на то, что синтаксически оператор инициации процесса близок к оператору обращения к подпрограмме, семантически они различаются достаточно сильно и время инициации процесса во много раз больше, чем время обращения к подпрограмме.

Для нормального завершения процессов обычно используются традиционные операторы типа STOP, END, RETURN.

Завершение вызванного процесса и всех порожденных им параллельных процессов происходит при достижении им оператора RETURN или END. Заметим, что хотя языковые конструкции завершения процесса вполне традиционны, их семантика сложнее, чем в последовательных языках программирования. В некоторых языках программирования операторы RETURN и END завершают поддерево процессов, а оператор STOP – всего графа процессов.

Иногда в языках программирования для порождения нескольких параллельных процессов (агентов) в одной точке используются операторы типа FORK и COBEGIN, а для объединения процессов (агентов) в одной точке – JOIN и COEND. Оператором FORK из одного процесса можно породить другие, и они будут выполняться параллельно до тех пор, пока либо вызывающий процесс не выполнит оператор JOIN, либо не закончится выполнение параллельных процессов. Существуют и более жесткие конструкции. Так конструкция COBEGIN, COEND в некоторых языках порождает все процессы, поименованные в конструкции COBEGIN, COEND в одной точке процесса, и завершает все эти процессы – в другой. Таким схемам параллельных вычислений в разных языках программирования соответствуют различные служебные слова и другие синтаксические конструкции, а не только упомянутые выше. Более того, один и тот же синтаксис в различных языках программирования может иметь различную семантику.

Повторим еще раз, что инициация процесса (агента), несмотря на его внешнее сходство с обращением к подпрограмме и процедуре, значительно сложнее. Поэтому элегантность гибких языковых конструкций может обернуться неэффективностью их реализации. В частности, инициация N параллельных процессов (агентов), объявленных в операторе языка, вовсе не означает, что все N процессов будут одновременно инициированы. Это означает только, что они будут поставлены в очередь к соответствующим процессорам, а начнут выполняться только тогда, когда эти процессоры освободятся. Введение сложных дисциплин выполнения процессов, указанных в операторах, потребует от операционной системы создания очередей с соответствующими дисциплинами управления.

Но и слишком большая жесткость может обратиться неудобством. Так, например при одной из реализации CONCURRENT PASCAL было введено ограничение на инициацию числа процессов, реализующих тело (описание) одного и того же процесса. Это привело, в частности, к тому, что операционные системы, написанные на этой реализации CONCURRENT PASCAL, могут иметь только фиксированное число процессов и мониторов. При необходимости увеличения числа одновременно выполняемых процессов и синхронизирующих мониторов в тех операционных системах их необходимо перекомпилировать.

Операторы типа CALL, FORK и COBEGIN, как уже было отмечено выше, ставят готовые к выполнению процессы в очередь, из которой они выбираются по мере освобождения вычислительных ресурсов.

В упоминавшемся уже языке Java поток также получает возможность создать и запустить другие потоки. В предыдущем примере поток BestDecision порождает потоки class ScenarioGeneration и class ScenarioEvaluation.

Для запуска потока вместо метода main можно использовать метод start. В этом случае запуск нового потока из порождающего его потока может быть описан следующим образом:

Thread, thread=new Thread ( );

thread, start ( );

Первая строка создает новый объект: поток new Thread, вторая строка будет запускать этот объект на выполнение. Запуск "повторим еще раз", будет произведен тогда, когда для этого будут созданы условия и появится возможность.

Каждый поток может находиться в одном из трех состояний: "поток создан, но еще не запущен", "поток выполняется", "поток в пассивном состоянии" ("замороженном" по терминологии рис. 1,*б*). Эти состояния эквивалентны состояниям агента рис. 1,*б*.

Чрезвычайно важным свойством потоков является асинхронность их выполнения. В процессе их выполнения они состязаются за ресурсы. Приводимый ниже пример [30] показывает как два процесса, борясь за ресурсы, состязаются друг с другом в скорости выполнения. Гонка охватывает только цикл for в каждом из потоков, которые печатают сообщения, показывающие на каком шаге гонки они находятся.

```
Race.java
class Racer1 extends Thread {
  public void run ( ) {
      for (int i=1; i<=10; ++i)
         System.out.println ("Racer1: step" +i);
  }
}
class Racer2 extends Thread {
  public void run ( ) {
      for (int i=1; i<=10; ++i)
         System.out.println ("Racer2: step" +i);
  }
}
public class Race {
  public static void main (String args [] ) {
     new Racer1().start();
     new Racer2().start();
  }
}
cmd>java Race
Racer1: step 1
Racer2: step 1
Racer1: step 2
Racer2: step 2
Racer1: step 3
Racer2: step 3
Racer1: step 4
Racer2: step 4
Racer1: step 5
Racer1: step 6
Racer1: step 7
Racer1: step 8
Racer2: step 5
Racer2: step 6
Racer2: step 7
Racer2: step 8
Racer2: step 9
Racer2: step 10
Racer1: step 9
```

Racer1: step 10

Обратите внимание: вначале оба процесса (агента) идут, как говорят на скачках, "ухо в ухо", затем первый процесс обгоняет второй, но в конце концов второй процесс заканчивает "скачку" первым. Если не использовать специальные средства, например, присваивая потокам приоритеты, очередность выполнения асинхронных параллельных процессов становится непредсказуемой.

Необходимо подчеркнуть, что инициация параллельного процесса одновременно является и эффективным средством синхронизации. Еще одним средством инициации параллельных процессов (асинхронно выполняющихся агентов) является вызов удаленной процедуры.

Языковая конструкция вызова удаленной процедуры, обычно аналогична вызову традиционной (последовательной) процедуры CALL.

Вызов удаленной процедуры выполняется следующим образом: входные аргументы посылаются соответствующему адресату и вызывающий процесс задерживается до тех пор, пока вызываемая процедура не будет выполнена и результаты не будут переданы вызывающему процессу в виде выходных параметров. Таким образом, вызов удаленной процедуры также является одним из средств синхронизации.

Заметим, что такой оператор CALL по своей семантике эквивалентен паре операторов SEND, RECEIVE (послать, получить).

Существуют два подхода для описания удаленной процедуры. В первом подходе удаленная процедура описывается как процедура в последовательных языках. Например,

REMOTE PROCEDURE <имя> (список параметров)
    Тело процедуры
END

Такая процедура выполняется как процесс. Этот процесс ожидает получения сообщения от вызывающего процесса, выполняет тело процедуры и возвращает выходные параметры.

Во втором подходе удаленная процедура рассматривается как оператор, который может быть помещен в любом месте процесса. Этот оператор может быть записан, например, в таком виде.

ACCEPT <имя> (список параметров) → тело процедуры.

Если оператор вызова готов к выполнению раньше, чем вызываемый процесс готов его принять, то выполнение вызывающего процесса задерживается, пока выдаваемый процесс не примет вызов.

Если оператор приема готов к выполнению раньше, чем появился соответствующий вызов, то этот вызов при его появлении может быть обработан немедленно, но пока не будет выдан ответ, вызываемый процесс будет задержан. Таким образом достигается синхронизация процессов, получившая название "рандеву", используемая в широко известном языке программирования АДА [38].

В тех случаях, когда в процессе имеется несколько операторов типа ACCEPT или AWAIT и к ним создаются очереди, возникает необходимость упорядочить выполнение этих операторов.

Для этой цели может быть использован оператор типа SELECT. Он может иметь вид:

SELECT
  [WHEN <булевское выражение> → <оператор ACCEPT>
      <последовательность операторов>]
      {OR [WHEN <булевское выражение> →]<оператор ACCEPT>
      [<последовательность операторов>]}
      [ELSE <последовательность операторов>]
  END SELECT;

Выполнение оператора SELECT производится следующим образом:

1. Вычисляются все булевские выражения, включенные в оператор SELECT. Каждый оператор ACCEPT, в котором булевское выражение равно "истина", помечается меткой готовности. Оператор ACCEPT, которому не предшествует булевское выражение, всегда помечен меткой готовности.

2. Оператор ACCEPT, помеченный меткой готовности, инициируется только тогда, когда другой процесс обращается к этому оператору. Если могут быть выбраны несколько операторов ACCEPT, то выбор производится случайным образом. Если ни один оператор ACCEPT не может быть выбран и в операторе SELECT есть конструкция ELSE, то выполняются операторы, входящие в конструкцию ELSE. Если конструкция ELSE отсутствует, то процесс ожидает, пока соответствующий оператор ACCEPT не будет выбран.

3. Если ни один оператор ACCEPT не помечен меткой готовности, и есть конструкция ELSE, то выполняются операторы, входящие в конструкцию ELSE. В противном случае возникает исключительная ситуация.

Таким образом, оператор ACCEPT обеспечивает процесс механизмом ожидания определенного события в другом процессе, а оператор SELECT обеспечивает процесс механизмом ожидания множества событий, порядок поступления которых непредсказуем.

В языке Java, как было показано выше, вызов удаленной процедуры решается принципиально иначе. Но конечно, проблема синхронизации остается.

Синхронизация в Jav'e может быть выполнена при помощи оператора wait. Например:

```
 synchronized void waitForCond () {
    while (check Condition () = = false ) {
       wait ();
       }
}
void someMethod {
    waitForCondition ();
     ...// продолжается с кода, для которого условие должно быть
          истинным//
}
```

Заметим, что с оператором Wait мы уже встречались в начале этого раздела. Его семантика была та же, что и в Jav'e.

В языке Java потоки имеют приоритеты. Все потоки начинают жизнь с приоритетом, равным приоритету потока, который их породил. Первоначальной установкой является NORM_PRIORITY. Имеются еще две установки приоритета в Thread: MAX_PRIORITY и MIN_PRIORITY.

Ключевое слово языка Java synchronized (синхронизация) дает возможность потоку получить исключительное право на управление методом, гарантирующее непрерывный доступ к нему (или к нескольким методам).

Добавление модификатора синхронизации к объявлению метода делает поток доступным методу так долго, как это ему необходимо для выполнения вычислений, т.е. до тех пор, пока он использует метод. Все остальные потоки, желающие получить доступ к методу должны ожидать окончания действия модификатора synchronized для этого потока. После этого доступ к синхронизированному методу получает следующий в очереди поток.

Таким образом, операторы синхронизации языков высокого уровня неявно реализуются механизмом семафора.

5. Организация обмена сообщениями между агентами (процессами)

Обмен сообщениями является основным механизмом передачи информации между агентами в распределенных системах. Возможны два режима обмена сообщениями.

Синхронный. В этом режиме при посылке сообщения агент-отправитель прекращает выполнение программы до получения ответа от агента-адресата. Такой режим снижает параллелизм работы системы.

Асинхронный. В этом режиме каждый активный агент располагает очередью сообщений. Когда сообщение посылается, агент-отправитель записывает информацию об отосланном сообщении в очередь ожидаемых ответов, продолжая выполнять программу (не ожидая пока ответ будет получен). Такой режим не снижает параллелизм работы. Рис. 6,*а* иллюстрирует синхронный режим, а рис. 6,*б* – асинхронный. Как видно из рис. 6 обмен сообщениями также является средством синхронизации.

Обычно в распределенных системах рассматриваются две модели обмена сообщениями.

Клиент-сервер. Сервер – всегда активный процесс, ожидающий запроса клиента. Обрабатывает его запрос и посылает ответ клиенту. Эта схема сейчас наиболее популярна. Архитектура WWW, как и архитектура других видов сервиса Интернет, построена по принципу клиент – сервер. Основной задачей программы-сервера является организация доступа к информации, хранящейся в компьютере, на котором эта программа функционирует в режиме ожидания запросов от программ-клиентов. Когда программе-клиенту необходимо получить некоторую информацию от сервера, она направляет к нему запрос. При достаточных правах доступа (это очень важно) между программами устанавливается соединение и сервер направляет ответ на запрос. После этого установленное соединение разрывается. При возникновении нового запроса процесс повторяется. Функционирование схемы клиент – сервер показано на рис. 7.

Клиент – клиент. В этой модели все процессы (или узлы сети) считаются равными. Обмен между ними происходит без промежуточного обращения к серверу. Функционирование схемы клиент – клиент показано на рис. 8.

В системах с раздельной памятью и в локальных сетях для обмена информацией требуются новые языковые средства и серьезная системная поддержка.

Особенность обмена информацией в таких системах заключается в том, что в обмене взаимодействуют, по крайней мере, две достаточно автономные стороны: отправитель и адресат. Причем адресат в момент получения сообщения может оказаться занятым и сразу же принять сообщения не может, а отправитель должен задержать выполнение своего процесса, чтобы убедиться, что его сообщение принято адресатом, а во многих случаях и дождаться ответа с результатами обработки адресатом его сообщения. Таким образом, при обмене информацией между асинхронными параллельными процессами в многопроцессорных системах с раздельной памятью и в локальных сетях всегда присутствуют элементы синхронизации. Передача сообщения всегда включает в себя, по крайней мере, два оператора: оператор в процессе, отправляющем сообщение, и оператор в процессе, принимающем сообщение. Поэтому в системе передачи сообщений важно, кто партнеры (это могут быть процессы, модули, задачи и т.д.), как устанавливается связь (статически или динамически), каков метод передачи данных (по параметрам, присваиванием и т.д.), что является объектом адресации (имя процесса, имя процедуры, имя входа, имя порта и т.д.), каковы условия приема сообщения (безусловное, только от определенного источника, в зависимости от состояния принимающего процесса и т.д.), что содержит ответ (подтверждение получения, результат обработки сообщения) и т.п.

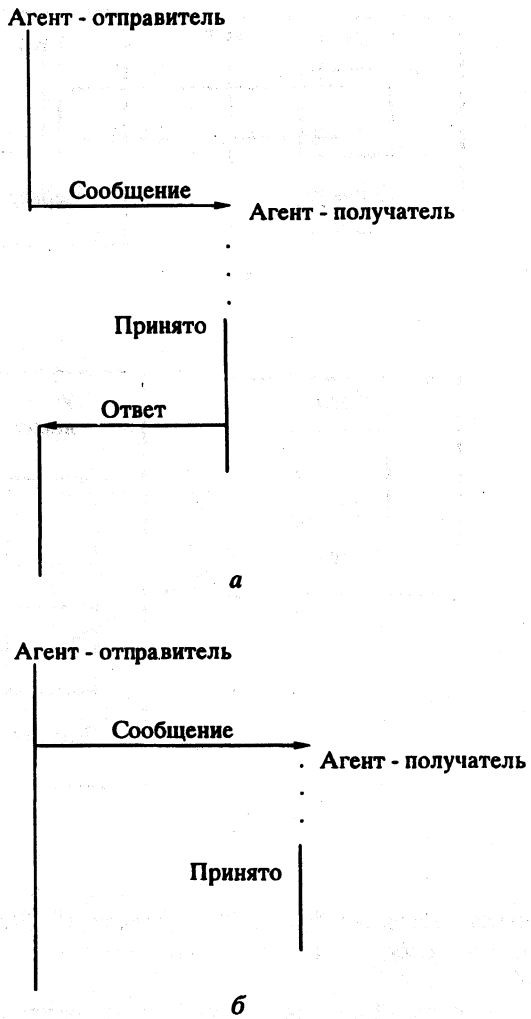


Рис. 6

Конечно, далеко не все эти факторы отражаются в языковых конструкциях. Часть определяется системными средствами, но так или иначе они находят свое отражение в системе программирования.

Простейшая структура обмена информацией между агентами, параллельно работающими на двух процессорах, показана на рис. 9.

Первый агент находится в состоянии "работа 1,1". Когда он достиг точки, в которой необходима синхронизация, он посылает агенту (состояние "послать 1") сообщение ("сообщение 1"), в котором содержится информация, необходимая для синхронизации. Послав сообщение, первый агент переходит в состояние ожидания ("работа 1,2") до получения ответа от второго агента. Пока первый агент находится в ожидании, "сообщение 1" принимается другим агентом. Если второй агент готов принять сообщение, т.е. находится в состоянии ожидания получения информации (сообщение "ждать 2"), то он принимает сообщение ("работа 2,1") и обрабатывает информацию, полученную в "сообщении 1". Результат этой обработки передается

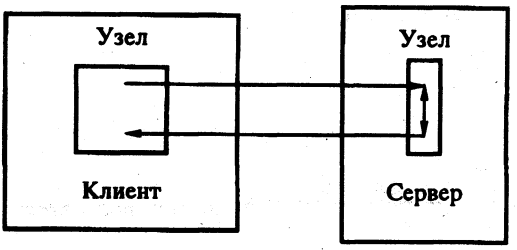


Рис. 7

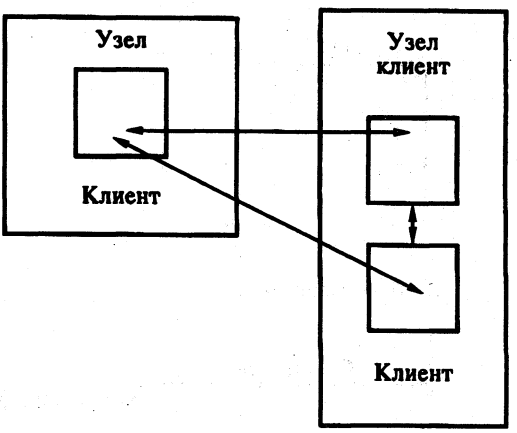


Рис. 8

первому агенту через состояния "ответ 2" и "ждать 1". После этого первый агент переходит в состояние "работа 1,1", чем и завершается синхронизация посредством обмена сообщений.

Итак, рассмотрен лучший случай – второй агент ожидал сообщения первого. Но на практике в большинстве случаев сообщение ждет того момента, когда агент, к которому оно послано, сможет его принять и обработать. Больше того, может образоваться очередь сообщений к какому-либо агенту от других агентов. И все агенты, пославшие сообщения, будут ожидать их обработки. Не рассматривая способы управления создавшимися очередями, подчеркнем, что ожидание агентом ответа на посланное сообщение является одной из серьезных причин, снижающих эффективность работы распределенных вычислительных систем.

Операторы посылки и приема содержат соответствующие служебные слова языка, адреса отправителя и адресата и список переменных, значение которых пересылается из одного процесса в другой. Ниже приводится пример одной из самых простых языковых конструкций посылки и приема сообщений.

SEND <список значений> TO <адресат-получатель>, где <список значений> содержит значение выражений в момент посылки сообщений, а <адресат-получатель> определяет адресата сообщения и оператор, который его отправил.

RECEIVE <список переменных> FROM <адресат-получатель>, где <список переменных> не требует комментариев, а <адресат-получатель> – адрес отправителя сообщения и оператор, которому оно направляется.

Приведенная языковая конструкция в таком виде встречается в языках программирования не часто, но сущность обмена сообщениями иллюстрирует хорошо.

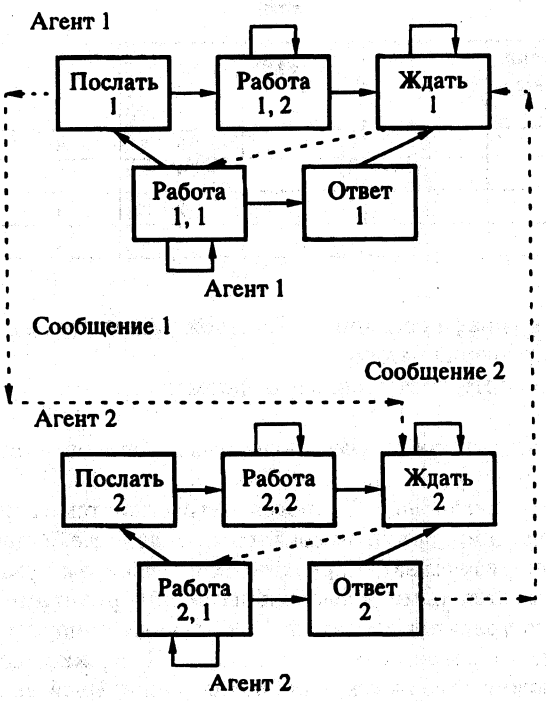


Рис. 9

На прием могут быть наложены условия, например, при помощи оператора WHEN L. Тогда сообщение будет принято только в том случае, если значение логической переменной L – истина, в противном случае оно ставится в очередь.

Функция оператора WHEN, определяющего условия приема сообщения, может быть расширена введением конструкции "приоритет". Оператор имеет вид

AWAIT BY (<выражение – приоритет>)

При выполнении оператора AWAIT вычисляется значение булевской переменной (или булевского выражения) L. Если истинно – сообщение принимается, если ложно – ставится в очередь в соответствии со значением выражения – приоритета. Конструкция BY не является обязательной.

6. Управление вычислительным процессом в многоагентных системах поддержки принятия решений

Для диспетчеризации процессов и агентов в распределенных системах необходимо:

– определить дисциплину диспетчеризации процессов и агентов, находящихся в каждом узле системы,

– определить дисциплину их мониторинга в распределенной системе.

А. Диспетчеризация задач в узле многоагентной системы. Поскольку задачи, агенты и процессы, реализуемые в многоагентной системе, тесно связаны между собой, то при их диспетчеризации, помимо традиционно оцениваемых характеристик, необходимо учитывать:

– локальность задачи, решаемой агентом или процессом (решается ли она только в данном узле, в нескольких узлах или всей системой);

Таблица 5

Уровни графа / Приоритеты	5	4	3	2	1	0
4	4	4	4	4	4	4
3	1	2	2	3	3	3
2	1	1	1	1	2	2
1	1	1	1	1	1	1

– к какому уровню графа решения глобальной задачи принадлежит данная задача и каково состояние решения задачи;

– срочность (важность поступившей информации);

– достоверность.

Один из возможных способов учета этих факторов – оценивать их через приоритет задачи.

Локальность (*глобальность*). В многоагентных системах может возникнуть необходимость различать приоритеты локальных задач, т.е. задач, решаемых только в данном узле, и приоритеты задач, решаемых в нескольких узлах. Будем называть их соответственно локальными и глобальными приоритетами. Может оказаться, что их нежелательно уравнивать, так как глобальные приоритеты задаются в верхних узлах иерархии, а локальные – в нижних. Задержка одного этапа решения в одном из узлов может задержать выполнение важнейшей задачи, решаемой всей сетью.

На значение приоритета должен влиять уровень работы в графе решения задачи, распределенной на решение по нескольким узлам. Чем больше работ зависит от данной работы, тем меньше времени она должна находиться в очереди, тем выше должен быть ее приоритет. Если исходить из предположения, что чем ниже в графе решения задачи находится такая работа, тем больше работ от нее зависит, то естественно повышать приоритет таких работ, внося корректировку в значение их приоритета. Скорректированное значение приоритета j-й работы, k-го приоритета удобно представить в табличной форме, например, в виде табл. 5. По вертикали на табл. 5 отложены приоритеты задач, а по горизонтали – уровни графа решения задач, на которых находятся j-е работы. В клетках таблицы указан приоритет задачи, учитываемый операционной системой при выборе задачи из очереди на решение. Задачи 4-го приоритета в нашем примере являются фоновыми и на их приоритет уровень работы в графе задачи влияния не оказывает.

Важность поступившей информации. В информационных системах и системах поддержки принятия решений, работающих в реальном времени, особое значение приобретает оценка важности поступившей информации, например, резкое повышение радиационного фона, информация о передислокации частей противника, данные о сбоях в системах управления повышенной опасности и т.д. СППР должны уметь распознавать такие сообщения и присваивать обрабатывающим их процессам высокие приоритеты.

Достоверность поступающей информации. Достоверность поступающей информации может оцениваться различными средствами. Например, в системах анализа и ликвидации последствий радиоактивного заражения сигналы о повышении уровня радиации, тем более резкого, могут повышать приоритеты задач, обрабатывающих эту информацию. Достоверность такой информации может определяться числом датчиков, сигнализирующих о повышении уровня радиации. Достоверность может оцениваться априорно: данные каждого источника заранее получают оценку достоверности от эксперта. Наконец, оценка достоверности информации, передавае-

Таблица 6

$i \setminus j$	1	2	3	...	n
2	3,0	–	2,1		1,4

мой каждым узлом СППР, может определяться на основе систематического анализа получаемой информации и оценки ее достоверности. Оценка информации может накапливаться в строке таблицы, аналогичной табл. 6, где индекс i обозначает номер агента, получающего информацию, а j – номер узла (или агента), передающего данные. В табл. 6 показана оценка 2-м агентом надежности данных, получаемых от других агентов и/или узлов. Оценка производится по трехбалльной шкале: информация достоверна (3), не вполне достоверна (2) и ложная (1). Балльность, конечно, может быть любой. Первоначально строка заполняется ЛПР в соответствии с его представлениями о надежности получаемых данных. После получения каждой новой информации агентом i от источника j, он производит оценку этой информации с учетом достоверности ранее полученных данных, например, методом сглаживания (берется среднее значение m последних оценок достоверности). Моментальный снимок таких оценок дан в табл. 6.

Таким образом, для узла многоагентной системы можно ввести следующие критерии диспетчеризации:

– традиционный (приоритет задачи или работы, объем занимаемой памяти, время решения, интенсивность ввода/вывода и т.д.);

– локальность (глобальность) задачи;

– важность поступившей информации;

– достоверность поступившей информации.

Правила повышения приоритета задачи в связи с повышением важности обрабатываемой ею информации и степени ее достоверности формулируются в соответствии с характером задачи и в операционной системе могут быть заложены в виде функций предпочтения или других алгоритмов. Задачи, получившие в результате таких оценок высшие приоритеты, будем называть задачами, находящимися в фокусе внимания [39].

Пусть A – множество задач и G – множество критериев, $A \cap G = 0$, $a \in A$, $g \in G$, $\delta \in [0,1]$ [40]. Для каждого критерия g сформируем функцию c_g, определяющую насколько задача a удовлетворяет критерию g,

$$c_g(a) = \begin{cases} \delta, \text{ если задача } a \text{ способствует выполнению критерия } g \\ \text{в степени } g; \\ 0 - \text{в противном случае} \end{cases}$$

и функцию d_g, определяющую насколько задача a препятствует (мешает) выполнению критерия g

$$d_g(a) = \begin{cases} \delta, \text{ если задача } a \text{ препятствует выполнению критерия } g \\ \text{в степени } g; \\ 0 - \text{в противном случае.} \end{cases}$$

Оценочная функция задачи, определяющая ее приоритет может иметь вид:

$$V(a) = \sum_{g \in G} (P_g[c_g(a) - d_g(a)]),$$

где P_g – "вес" g-го критерия.

Оценочная функция $V(a)$ хорошо проясняет семантику нахождения задач, входящих в фокус внимания, но определить значения δ в функциях $c_g(a)$ и $d_g(a)$, как правило, трудно.

При разработке операционных систем было придумано много эвристических алгоритмов для нахождения приоритета задач, находящихся в очереди, их ранжирования и определения порядка их запуска на решение. В большинстве случаев эти алгоритмы сводились к оценке сумм вида $\sum p_i q_i$, где q_i – критериальная оценка значения физического параметра по i-му критерию, а p_i – "вес" (значимость) i-го критерия. Основываясь на этом опыте, для ранжирования задач можно предложить следующий подход.

Нечеткое отношение предпочтения по j-му критерию $p_j(k,\ell)$ для пары альтернатив (A_k, A_ℓ) определим функцией принадлежности:

$$(1) \qquad p_j(k,\ell) = \begin{cases} \dfrac{r_{kj} - r_{\ell j}}{m_j}, & \text{если } r_{kj} > r_{\ell j}; \\ 0 - \text{в противном случае}, \end{cases}$$

где m_j – балльность шкалы оценок по j-му критерию, а r_{kj} и $r_{\ell j}$ – нечеткие переменные, характеризующие оценки k-й и ℓ-й альтернативы по j-му критерию.

Нечеткое отношение предпочтения по паре альтернатив (A_k, A_ℓ) определим функцией:

$$(2) \qquad P(k,\ell) = \sum_{j=1}^{J} k_j p_j(k,\ell) = \sum_{j=1}^{J} k_j \begin{cases} \dfrac{r_{kj} - r_{\ell j}}{m_j}, & \text{если } r_{kj} > r_{\ell j}; \\ 0 - \text{в противном случае}, \end{cases}$$

где k_j – "нормированный" вес (значимость) j-го критерия.

Заметим, что (2) интерпретируется как степень согласия ЛПР с тем, что k предпочтительнее ℓ, а функция $P(\ell,k)$ – функция несогласия с этим утверждением.

Нечеткое отношение строгого предпочтения альтернативы A_k над альтернативой A_ℓ определим функцией $\mu_D(k,\ell)$, характеризующей интенсивность доминирования (предпочтительности).

$$(3) \qquad \mu_D(k,\ell) = \begin{cases} P(k,\ell) - P(\ell,k), & \text{если } P(k,\ell) > P(\ell,k); \\ 0 - \text{в противном случае}. \end{cases}$$

Функция $\mu_D(k,\ell)$ должна обладать следующими свойствами [41]:

1. $\mu_D(k,\ell)$ возрастает с увеличением превосходства альтернативы A_k над альтернативой A_ℓ, так в частности, $\mu_D(k,\ell)$ – неубывающая функция от r_{kj}, $\forall_j$ и невозрастающая функция от $r_{\ell j}$, $\forall_j$;

2. $\mu_D(k,\ell) = 1$ означает безусловное превосходство альтернативы A_k над альтернативой A_ℓ;

3. $\mu_D(k,\ell) = 0$ означает безусловное отсутствие превосходства альтернативы A_k над альтернативой A_ℓ или полное отсутствие аргументов в пользу превосходства одной альтернативы над другой.

Естественно определить отношение недоминирования $\mu_{ND}(k,\ell)$ как дополнение к $\mu_D(k,\ell)$

$$(4) \qquad \mu_{ND}(k,\ell) = 1 - \mu_D(k,\ell).$$

Множество недоминируемых альтернатив $\mu_D^*(A_k)$ получается следующим очевидным образом [42]:

$$(5) \qquad \mu_D^*(A_k) = \min_{\substack{\ell=1,\ldots,m \\ \ell \neq k}} \mu_{ND}(\ell,k) = \min[1 - \mu_D(\ell,k)] =$$

$$= 1 - \max \mu_D(\ell,k) = 1 - \max[P(\ell,k) - P(k,\ell)].$$

Таблица 7

№ п/п	Критерий	Обозначения	Веса критериев
1	локальность	лок.	не очень важный
2	важность	важ.	очень важный
3	достоверность	дост.	важный

Лучшая альтернатива соответствует условию:

$$\mu_D^*(A_k^*) = \max_{k=1,\dots,m} \mu_D^*(A_k) = 1 - \min\{ \max_{\substack{k=1,\dots,m \\ \ell=1,\dots,m \\ \ell \neq k}} [P(\ell,k) - P(k,\ell)]\}. \tag{6}$$

Алгоритм ранжирования альтернатив может иметь следующий вид:

1. Инициализация задачи: задание базовых шкал, определение по ним критериальных оценок альтернатив, которые сводятся в матрицу оценок E, задание вектора V "весов" критериев.
2. Вычисление значений $p_j(k,\ell)$ по каждому критерию для каждой пары альтернатив по формуле (1).
3. Вычисление каждой пары альтернатив $P(k,\ell)$, $\forall_k$, $\forall_\ell$, по формуле (2).
4. Вычисление отношений доминирования по формуле (3).
5. Вычисление отношений недоминирования по формуле (4).
6. Вычисление интенсивности доминирования каждой альтернативы по формуле (5).
7. Определение лучшей альтернативы по формуле (6). Ранжирование альтернатив.

Рассмотренный выше подход является развитием методов, изложенных в [41–46], вводящих функцию согласия (concordance) и несогласия (discoradance) и пороговые значения для определения отношений эквивалентности, предпочтения и значительного предпочтения. Использование лингвистических переменных или балльных оценок, полученных либо с помощью базовых шкал [47], либо непосредственно от пользователя позволяют отказаться от трудно определяемых пороговых значений.

Рассмотрим иллюстративный пример.

1. Пусть в очереди на решение в узле многоагентной системы стоят четыре задачи. Для нахождения фокуса внимания они оцениваются по трем критериям: достоверность полученной информации, ее важность и локальность задачи (см. табл. 7). Традиционные оценки приоритета, применяемые в традиционных операционных системах при определении фокуса внимания, в нашем примере не учитывались.

Диспетчер агента преобразует их из лингвистического вида в балльный (преобразование показано в матрице E и векторе V)

$$E = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{array} \begin{array}{c} \begin{array}{ccc} K_{\text{дост.}} & K_{\text{важ.}} & K_{\text{лок.}} \end{array} \\ \begin{bmatrix} \text{удовл.} & \text{удовл.+} & \text{плохо} \\ \text{хорошо} & \text{удовл.} & \text{оч. плохо+} \\ \text{плохо+} & \text{хорошо+} & \text{хорошо} \\ \text{удовл.+} & \text{удовл.+} & \text{отлично} \end{bmatrix} \end{array} = \begin{bmatrix} 5 & 6 & 3 \\ 7 & 5 & 2 \\ 4 & 8 & 7 \\ 6 & 6 & 9 \end{bmatrix}.$$

Веса критериев будем оценивать по пятибалльной шкале. Оч. важно – 5, важно – 4, не очень важно – 3 и т.д.

$$V = \begin{array}{c} \begin{array}{ccc} \overline{K}_{\text{дост.}} & \overline{K}_{\text{важ.}} & \overline{K}_{\text{лок.}} \end{array} \\ [\text{важный} \quad \text{оч. важный} \quad \text{не оч. важный}] \end{array} = [4,\ 5,\ 3].$$

2. По формуле (1) и матрице E диспетчер находит $p_j(k,\ell)$, $\forall_k$, $\forall_\ell$.

$$p_1(1,2)=0,\quad p_2(1,2)=\frac{r_{12}-r_{22}}{m_1}=\frac{6-5}{10}=0{,}1,\quad p_3(1,2)=0{,}1.$$

Таким образом вектор $p_j(1,2)=[0\ 0{,}1\ 0{,}1]$ $p_j(1,3)=[0{,}1\ 0\ 0]$ $p_j(1,4)=[0\ 0\ 0]$ аналогично

$$p_j(2,*)=\begin{matrix}\\(2,1)\\(2,3)\\(2,4)\end{matrix}\begin{matrix}j=\\\\\\\end{matrix}\begin{matrix}1&2&3\\\\\\\end{matrix}\begin{bmatrix}0{,}1&0&0\\0{,}3&0&0\\0{,}1&0&0\end{bmatrix}\qquad p_j(3,*)=\begin{matrix}\\(3,1)\\(3,2)\\(3,4)\end{matrix}\begin{matrix}j=\\\\\\\end{matrix}\begin{matrix}1&2&3\\\\\\\end{matrix}\begin{bmatrix}0&0{,}2&0{,}4\\0&0{,}3&0{,}5\\0&0{,}2&0\end{bmatrix},$$

$$p_j(4,*)=\begin{matrix}\\(4,1)\\(4,2)\\(4,3)\end{matrix}\begin{matrix}j=\\\\\\\end{matrix}\begin{matrix}1&2&3\\\\\\\end{matrix}\begin{bmatrix}0{,}1&0&0{,}3\\0&0{,}1&0{,}7\\0{,}2&0&0{,}2\end{bmatrix}.$$

3. Диспетчер производит нормирование значений $\overline{K}_j$ вектора V и по формуле (6) находит значения $p_j(k,\ell)$, $\forall_k$, $\forall_\ell$

$$\sum_{j=1}^{5}\overline{k}_j=15,\quad k_1=4/15=0{,}26,\quad k_2=5/15\doteq 0{,}33,\quad k_3=3/15=0{,}2,$$

$$P(1,2)=\sum_{j=1}^{3}k_jp_j(1,2)=0{,}26*0+0{,}33*0{,}1+0{,}2*0{,}1=0{,}056,$$

$$P(1,3)=0{,}026,\quad P(1,4)=0,$$

аналогично находим матрицу $P(k,\ell)$

$$P(k,\ell)=\begin{matrix}\\a_1\\a_2\\a_3\\a_4\end{matrix}\begin{matrix}a_1&a_2&a_3&a_4\\\\\\\end{matrix}\begin{bmatrix}0&0{,}056&0{,}026&0\\0{,}052&0&0{,}078&0\\0{,}146&0{,}199&0&0{,}66\\0{,}146&0{,}206&0{,}092&0\end{bmatrix}.$$

4. Диспетчер находит матрицу $\mu_D(k,\ell)$ по формуле (3)

$$\mu_D(k,\ell)=\begin{matrix}\\a_1\\a_2\\a_3\\a_4\end{matrix}\begin{matrix}a_1&a_2&a_3&a_4\\\\\\\end{matrix}\begin{bmatrix}0&0{,}04&0&0\\0&0&0&0\\0{,}12&0{,}121&0&0{,}04\\0{,}146&0{,}206&0{,}026&0\end{bmatrix}.$$

5. Диспетчер находит матрицу $\mu_{ND}(k,\ell)$ по формуле (4)

$$\mu_{ND}(k,\ell)=\begin{matrix}\\a_1\\a_2\\a_3\\a_4\end{matrix}\begin{matrix}a_1&a_2&a_3&a_4\\\\\\\end{matrix}\begin{bmatrix}1&0{,}96&1&1\\1&1&1&1\\0{,}88&0{,}879&1&1\\0{,}854&0{,}794&0{,}974&1\end{bmatrix}.$$

6. Диспетчер вычисляет интенсивность доминирования каждой альтернативы по формуле (5).

$$\mu_D^*(a_1)=\min\mu_{ND}(2,1),\mu_{ND}(3,1),\mu_{ND}(4,1)=\min[1,\ 0{,}88\ 0{,}854]=0{,}854;$$

$$\mu_D^*(a_2)=0{,}794,\quad \mu_D^*(a_3)=0{,}974,\quad \mu_D^*(a_4)=1.$$

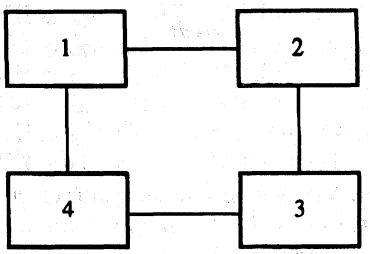


Рис. 10

7. Диспетчер определяет лучшую альтернативу по формуле (6)

$$\mu_D^*(a_k^*) = \max_K \mu_D^*(a_K) = a_4 = 1,$$

где a_4 должна принадлежать фокусу внимания.

Все четыре варианта могут быть ранжированы: $a_4 > a_3 > a_1 > a_2$.

Таким образом диспетчеризация задач в узле или агента многоагентной системы может быть сведена к традиционным приоритетным методам с рассмотренным выше расширением набора приоритетов. Естественно, методы вычисления приоритетов могут быть различными.

В. Мониторинг вычислительного процесса многоагентной системы. *Распределение агентов по процессорам.* Первая задача мониторинга – предварительное распределение агентов по процессорам. Решению этой задачи посвящено много работ [49, 50]. Рассмотрим подход, предложенный в [51], достоинством которого является хорошее прояснение сущности проблемы.

В качестве критерия выбрана минимизация времени обмена информации между процессами. Обычно число процессов M больше числа доступных процессоров N. Поэтому при распределении процессов (агентов) по процессорам необходимо избегать перегрузки процессоров, чтобы избежать блокировки или задержки выполнения параллельных процессов, кроме того процессы, интенсивно обменивающиеся информацией друг с другом, должны быть по возможности на минимальном "расстоянии" друг от друга, т.е. непосредственно связаны друг с другом, или по крайней мере, иметь небольшое число промежуточных узлов, через которые пересылаются сообщения.

Пусть c_{ij} – лингвистическая переменная, обозначающая интенсивность пересылки сообщений от процесса i к процессу j и обратно, т.е. $c_{ij} = c_{ji}$ и a_{ij} – лингвистическая переменная, характеризующая эффективность метода назначения процесса i на процессор j. Она определяется лингвистическими переменными "low" и "high" (возможны и другие оценки).

Метод распределения задач по процессорам поясним на простом примере. Для простоты будем считать, что необходимо на четырех процессорах разместить четыре процесса. Процессоры расположены так, как показано на рис. 10. Поскольку связи в сети симметричны, первый процесс может быть назначен на произвольный процессор: назначим его на первый процессор. Теперь фактически интересны те a_{ij}, у которых $j = 3$, т.е. те, которые обращаются к процессу, находящемуся на третьем процессоре. Поскольку на первый процессор назначение уже произошло, интерес представляют значения лингвистических переменных a_{23}, a_{33} и a_{43}, т.е. эффективность назначения процессов на процессор 3. Ясно, что после назначения некоторого процесса на процессор 3, оставшиеся два процесса могут быть назначены на процессоры 2 и 4 произвольно.

Ниже приводится множество правил, использующих лингвистические переменные a_{23}, a_{33} и a_{43}

R_1 – IF c_{12} is LOW AND c_{34} is LOW THEN a_{23} is HIGH
R_2 – IF c_{12} is HIGH AND c_{34} is HIGH THEN a_{23} is LOW
R_3 – IF c_{13} is LOW AND c_{24} is LOW THEN a_{23} is HIGH
R_4 – IF c_{13} is HIGH AND c_{24} is HIGH THEN a_{33} is LOW
R_5 – IF c_{14} is LOW AND c_{23} is LOW THEN a_{43} is HIGH
R_6 – IF c_{14} is HIGH AND c_{23} is HIGH THEN a_{43} is LOW

Они реализуют принцип: чем выше интенсивность обмена, тем "ближе" должны быть процессоры. Идеальный случай, чтобы они были в непосредственной близости.

Легко видеть, что каждое назначение определяется правилами R_1 – R_2, R_1 – R_2, R_5 – R_6. Правила R_1 и R_2 оценивают эффективность назначения процесса 2 на процессор 3, определяемого значением переменной a_{23}. Фактически оценивается назначение процесса 2 на процессор 3. Тогда процессы 3 и 4 должны быть назначены на процессоры 2 и 4 (неважно в каком порядке). Эффективность такого назначения, естественно, зависит от характера обмена информацией всех четырех процессов. Это можно сформулировать следующим образом:

– чем ниже c_{12} и c_{34}, тем выше a_{23} (из правила R_1);

– чем выше c_{12} и c_{34}, тем ниже a_{23} (из правила R_2).

Аналогично формулируются правила R_3 – R_4 и R_1 – R_6. В зависимости от того, какое из значений a_{23}, a_{33} и a_{43} окажется больше, определяется процесс (2, 3 или 4), назначаемый на процессор 3.

Конечно, значения лингвистических переменных не должны ограничиваться только "LOW" и "HIGH". Могут быть "MEDIUM", "VERY HIGH" и т.д.

Напомним, что c_{ij} и a_{ij} – лингвистические переменные и их значения определяются степенью принадлежности к определяемым ими понятиям.

В тех случаях, когда процессов больше, чем процессоров (типичный случай), необходимо учитывать:

– c_{ij} между процессами, уже назначенными на процессоры, и теми, которые должны быть назначены;

– c_{ij} между процессами, чье назначение на некоторые процессоры может быть изменено;

– суммарные значения c_{ij} процессов, уже назначенных на определенный процессор и тех, которые могут быть назначены.

Необходимо заметить, что с усложнением архитектуры сети, например с использованием тороидной структуры, и увеличением числа процессов на каждый процессор структура правил, оценивающих эффективность назначения процесса на процессор, сильно усложняется.

Правда, во многих случаях ввода начальные условия и условия назначения процессов на процессоры можно генерировать автоматически.

Управление вычислительным процессом в многоагентных системах. Теперь перейдем к управлению вычислительным процессом. Для решения этой задачи введем понятие единицы загрузки узла (или агента) многоагентной системы, позволяющее оценить степень загрузки каждого узла, и оценим вероятность его загрузки, после чего рассмотрим собственно алгоритмы мониторинга.

Единица загрузки узла. Один из возможных подходов для определения необходимых вычислительных ресурсов каждого агента или процесса – это введение понятия единицы загрузки узла сети (или процессора в узле сети).

Единица загрузки – это сумма оценок ресурсов различного вида, выделяемых для выполнения задачи. К таким ресурсам можно отнести время работы вычислительного процессора (T'), число операций ввода/вывода (I), объем оперативной памяти (S'). В зависимости от специфики системы возможны другие составляющие, влияющие на оценку загрузки системы.

Единица процессорного времени равна среднему времени выполнения данной моделью процессора некоторого фиксированного числа операций.

Единица ввода/вывода есть время выполнения одной операции ввода/вывода при вводе/выводе не больше некоторого фиксированного объема информации.

Единица оперативной памяти равна некоторому фиксированному объему оперативной памяти, используемому за единицу времени.

Тогда число единиц обслуживания для задачи i за время T_i при заданном числе операций ввода/вывода I_i и требуемом объеме памяти S_i

$$X_i(T_i, I_i, S_i) = \alpha \frac{T_i}{T'^j} + \beta I_i + \gamma \frac{S_i}{S'^j}, \tag{7}$$

где α, β, γ – коэффициенты, выбираемые в зависимости от характеристик вычислительной машины в узле сети. Изменяя коэффициенты α, β, γ можно менять "веса" каждой составляющей единицы обслуживания. Надо отметить, что подобный подход для оценки загруженности процессора применялся и в однопроцессорных системах, например, в OS/V2 для вычислительной машины IBM 370.

Введение понятия "единицы загрузки", определенное выше или каким-нибудь другим образом, дает возможность оценки – пусть достаточно грубо – степени загрузки каждого узла распределенной системы поддержки принятия решений и всей системы в целом. Максимальную загрузку узла, как обычно, будем оценивать за некоторый интервал времени T, при некотором числе операций ввода/вывода I за этот интервал, используя весь объем оперативной памяти S,

$$X \max(T, I, S) = \alpha \frac{T}{T'^j} + \beta I + \gamma \frac{S}{S'^j}.$$

Число единиц обслуживания для каждой задачи определяется по соотношению (7), а необходимое число единиц обслуживания для выполнения заданного набора задач за время T равно:

$$\sum_i^N X_i(T_i, I_i, S_i), \quad \sum_i T_i < T,$$

где N – число задач.

Аналогично, загрузка всей сети равна:

$$\sum_i^J \sum_i^N X_i(T_i, I_i, S_i), \quad \sum_i^J \sum_i^N T_i < T * J,$$

где J – число узлов.

Заметим, что T_i для каждой задачи будет меняться в зависимости от вычислительной машины, на которой она реализуется. В соответствии с этим будет меняться и X_i, однако, зная производительность "базовой" системы, для которой определялось T_i пользователем, и производительность машины, на которой задача может быть реализована, система автоматически производит необходимый пересчет.

Стохастическая оценка загрузки узла системы. Рассмотрим математическую модель, позволяющую хотя и достаточно грубо, но все же оценить вероятность того, что процессор в узле сети будет простаивать, т.е. степень его загрузки [52].

Будем считать, что длительность обмена информацией между каждой парой узлов сети является случайной величиной, имеющей экспоненциальное распределение с параметром λ. Это означает, что если узел передает или принимает информацию, то вероятность того, что он закончит передавать информацию до момента $t + h$, равна $P_1(h) = \lambda h e^{-\lambda h}$ и, в частности, при малых h имеет место $P_1(h) = \lambda h + 0(h)$.

Закончив сеанс обмена информацией, процессоры могут сразу начать следующий сеанс.

Пусть обработка процессором единицы загрузки занимает случайное время, распределенное по экспоненциальному закону с параметром μ. Это означает в случае работы процессора в момент t, что вероятность переработки данной единицы загрузки до момента $t+h$ равна при малых значениях h

$$P(h) = \mu h + 0(h).$$

Величины $1/\lambda$ и $1/\mu$ равны соответственно среднему времени обработки каналом и процессором единицы загрузки.

Процессор узла сети может находиться в одном из следующих состояний: S_0 – процессор простаивает, а все каналы работают на обмен; S_1 – один канал системы ожидает окончания обработки его информации, процессор узла сети работает, S_n – n каналов стоят в очереди на обработку, процессор работает; S_N – все N каналов ожидают в очереди на обработку, процессор работает.

Переход из состояния S_n в состояние S_{n+1} ($n+1$ каналы стоят в очереди на обработку) происходит по окончании передачи информации одним из каналов системы.

Переход из состояния S_n в состояние S_{n-1} означает, что процессор окончил переработку очередной части информации. Таким образом получается случайный процесс переходов из одного состояния в другое. Этот процесс можно характеризовать вероятностями $P_n(t)$ того, что в момент времени t система находится в состоянии S_n.

Найдем дифференциальные уравнения, которым удовлетворяют вероятности $P_n(t)$.

Вероятность того, что в момент $t+h$ будет состояние S_0, т.е. процессор будет простаивать, равна сумме вероятностей двух событий:

1. В момент t процессор простаивал и за время h ни один из каналов системы не закончил с ним обмена:

$$P_{00}(t+h) = P(t)(1 - N\lambda h) + 0(h).$$

2. В момент t процессор работал и за время h окончил работу, а каналы системы не закончили с ним обмен:

$$P_{10}(t+h) = P_1(t)(1 - N\lambda h)\mu h + 0(h) = P_1(t)\mu h + 0(h),$$

следовательно,

$$P_0(t+h) = P_0(t) - P_0(t)N\lambda h + \mu h P_1(t) + 0(h).$$

Перенеся $P_0(t)$ налево, разделив на h и перейдя к пределу, получим

$$P_0'(t) = -N\lambda P_0(t) + \mu P_1(t).$$

Учитывая, что переход в момент $t+h$ в состояние S_n может произойти как из состояния S_{n-1}, так и из состояний S_n и S_{n+1}, получаем, что эти вероятности удовлетворяют дифференциальным уравнениям [45]

$$\begin{aligned}
&P_0'(t) = -N\lambda P(t) + \mu P_1(t);\\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
&P_n'(t) = -\{(N-n)\lambda + \mu\}P_n(t) + (N-n+1)\lambda P_{n-1}(t) + \mu P_{n+1}(t);\\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
&P_N'(t) = -\mu P_N(t) + \lambda P_{N-1}(t).
\end{aligned} \tag{8}$$

В условиях стационарного режима работы процессора производные будут равны нулю и вероятности $P_0, P_1, \ldots, P_n, P_N$ не будут зависеть от момента времени t и удовлетворяют соотношениям, полученным из (8):

$$N\lambda P = \mu P_1;$$
$$\cdots\cdots\cdots\cdots$$
$$\{(N-n)\lambda + \mu\}P_n = (N-n+1)\lambda P_{n-1} + \mu P_{n+1};$$
$$\cdots\cdots\cdots\cdots$$
$$\mu P_N = \lambda P_{N-1}.$$

Отсюда получим рекуррентную формулу для расчета P_n:

$$(N-n)\lambda P_n = \mu P_{n+1}.$$

Раскрывая рекуррентную формулу, получаем:

$$P_{N-m} = \frac{1}{m!}\left(\frac{\mu}{\lambda}\right)^m P_N.$$

Вероятность P_N в силу того, что $\sum\limits_{m=1}^{N} P_m = 1$, равна

$$P_N = \frac{1}{1 + \sum\limits_{k=1}^{N} \frac{1}{k!}\left(\frac{\mu}{\lambda}\right)^k}.$$

Отсюда

$$P_{N-m} = \frac{\frac{1}{m!}\left(\frac{\mu}{\lambda}\right)^m}{\sum\limits_{k=0}^{N} \frac{1}{k!}\left(\frac{\mu}{\lambda}\right)^k}.$$

Таким образом, вероятность того, что процессор будет простаивать:

$$P_0 = \frac{\frac{1}{N!}\left(\frac{\mu}{\lambda}\right)^N}{\sum\limits_{k=0}^{N} \frac{1}{k!}\left(\frac{\mu}{\lambda}\right)^k}. \tag{9}$$

Значение P_0 позволяет оценить возможность процессора выполнить дополнительно вновь поступившую работу.

Передача информации и передача "полномочий". Оценка загрузки процессора или узла сети чрезвычайно важна, так как она позволяет использовать не загруженные или слабо загруженные процессоры и/или узлы для решения задач "чужих" агентов и процессов. Выше уже отмечалось, что объектно-ориентированное программирование существенно облегчает процесс "передачи полномочий" агента одного процессора и/или узла другому. На рис. 11 [8] показана схема "передачи полномочий". Сплошной линией показан физический поток передачи информации, штрихпунктиром – логический (так, как он представляется пользователю).

Одна из особенностей распределенных вычислительных систем заключается в том, что в них, как правило, осуществляется распределенное, а не централизованное управление вычислительным процессом.

Поэтому при организации управления вычислительным процессом каждый узел сети обязан представлять другим узлам информацию о своих вычислительных возможностях и своей загрузке. Для этого в каждом узле системы создается таблица загрузки узла, на которой указывается:

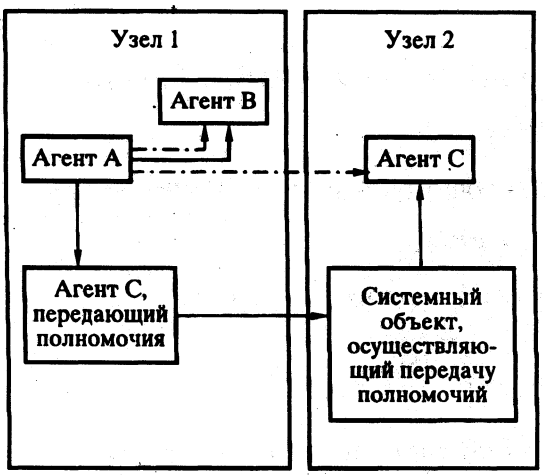


Рис. 11

– число единиц загрузки, которые может выполнить узел (вычислительная машина) сети – $X^{\max}$;

– число единиц загрузки, необходимых для решения задач, выполняемых в узле (вычислительной машине) сети – X'. Величины $X^{\max}$ и X' могут быть определены аналогично тому, как было рассмотрено в разделе "Единицы загрузки";

– вероятность P_0 того, что узел не будет загружен в интервале $t+h_0$ может быть определена по формуле (9);

– приоритеты задач, которыми загружен узел и которые стоят в очереди, могут быть определены в соответствии с разделом 6, А. Эта информация может располагаться на "доске объявлений", как показано на рис. 10.

В операционных системах до сих пор сохранилась устойчивая тенденция сочетания управления задачами, носящих регулярный характер (например, опрос датчиков и обработка результатов) и нерегулярных работ, запускаемых другой задачей, оператором, пользователем или инициативным датчиком. Периодически выполняемые задачи, реализующие регулярные функции, как правило, статически распределены по процессорам, но функции могут иметь разные приоритеты и при перегрузке процессоров более приоритетными задачами, выполнение некоторых функций может пропускаться или передаваться на менее загруженные узлы.

Для систем, работающих в реальном масштабе времени, например, СППР возникает проблема диспетчеризации задач таким образом, чтобы время отклика на поступившую информацию не превышало директивных сроков. Каждую i-ю задачу будем характеризовать тремя параметрами: временем счета T_i, моментом поступления t_i' и директивным сроком t_i''. В большинстве случаев момент наступления директивного срока отсчитывается от момента t_i', хотя, конечно, это может быть и астрономическое время. Для периодически решаемых задач обычно указывается еще и период τ_i поступления заявки на выполнение задачи. τ_i может быть использован для нахождения ближайших t_i'.

Поскольку в системы поддержки принятия решений информация, требующая немедленной обработки, может поступать в непредсказуемые моменты времени, в них необходимо использовать методы динамического планирования решения задач.

В настоящее время разработано достаточно большое число алгоритмов так или иначе реализующих дисциплины динамического планирования для задач, поступающих в непредсказуемые моменты времени. Рассмотрим возможную идеологию построения таких алгоритмов.

При диспетчеризации задач в распределенных системах сохраняется традиционное требование обеспечения достаточно быстрого выполнения задач реального времени с высокими приоритетами, но при этом возникает новая возможность: попытаться передать часть задач на решение в другие узлы распределенной системы. Рассмотрим подход, реализующий эту возможность.

Пусть в момент t_i' в систему поступают заявки на решение i-х задач Z_i. Они имеют приоритеты α_i, времена решения T_i и директивные сроки t_i'' $(T_i < t_i'' - t_i')$. Эта информация хранится в паспортах задач.

Необходимо проверить возможность выполнения этих n вновь поступивших задач.

Планирование осуществляется до окончания максимального директивного срока задачи из стоящих в очереди на решение задач. В тех случаях, когда в директивные сроки все задачи выполнены быть не могут, может быть использовано правило, обеспечивающее выполнение в директивные сроки задач с максимальным суммарным приоритетом. При этом учитывается число единиц загрузки X_i, требуемых i-й задачей и величина P_0 для данного узла. Идея алгоритма может быть сформулирована следующим образом [54]. Ищется

$$\sum_{i=1}^{n} \alpha_i Z_i \to \max$$

при ограничениях

$$Z_i = \begin{cases} 1, & \text{если } t_i'' \geqslant t_i' + T_i, \\ 0, & \text{если } t_i'' < t_i' + T_i, \end{cases}$$

где $i = \overline{1, n}$,

$$\sum_{i=1}^{n} X_i Z_i \leqslant X^{\max} - X, \quad P_0 > \text{const}.$$

P_0 может быть найдено в соответствии с (9), а const определяет некоторый порог вероятности того, что процессор будет свободен от выполнения текущих задач и сможет выполнить вновь поступившие.

Для тех задач, для которых $Z_i = 0$, может быть сделана попытка переслать реализующие их агенты для выполнения на другие процессоры.

Если все задачи выполняются в директивные сроки, то $Z_i = 1$ для $i = 1, 2, \ldots, n$.

Такой подход позволяет четко определить задачи, которые не могут быть решены в директивные сроки и должны быть реализованы в других узлах системы. (Если есть узлы, загруженные неполностью или вероятность загрузки которых до некоторого момента $t + h$ низкая.)

Исходя из этих данных, диспетчер узла сети определяет, ставит ли он поступившую задачу в очередь на выполнение или ищет незагруженный узел сети, который может ее выполнить.

Диспетчер узла сети периодически просматривает очередь задач и определяет, могут ли стоящие в очереди задачи быть выполнены в директивные сроки или они должны быть переданы на выполнение в другие узлы.

Динамика управления вычислительным процессом сети. Для возможности приема и передачи работ в распределенной системе должно поддерживаться несколько стандартных функций приема и передачи сообщений. Эти функции выполняются следующими системными процессами [55]:

– маршрутизатором, который обрабатывает все сообщения, для которых данный узел является источником, адресатом или промежуточным узлом на пути передач сообщений;

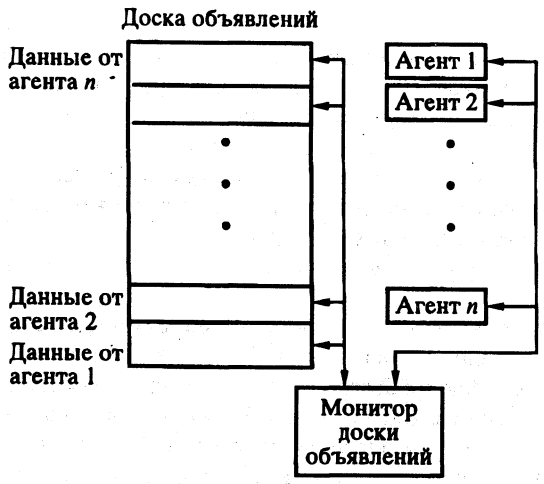


Рис. 12

– администратором данных, который управляет запросами данных от прикладных процессов;

– монитором прикладных процессов, который обеспечивает интерфейс между прикладными процессами, выполняющими требуемую обработку данных, и администратором данных;

– диспетчером задач, который обрабатывает все запросы задач, и определяет агенты и процессы, которые должны быть активизированы для выполнения задач.

При диспетчеризации процессов (агентов) обычно используются следующие типы правил [56, 57]:

– информационные правила, определяющие, где найти и куда переслать информацию, необходимую для принятия решений;

– правила пересылки, определяющие необходимость перемещения процесса (агента) с одного процессора на другой и, в случае необходимости, момент пересылки;

– правила выбора, определяющие процесс (агента), подлежащий пересылке;

– правила размещения, определяющие процессор, с которого и/или на который пересылается процесс (агент).

Вся необходимая информация может быть расположена на "доске объявлений", как показано на рис. 12.

Инициируя новую задачу, диспетчер задач должен учитывать, что возрастание количества задач увеличивает обмен сообщениями в распределенной системе, особенно это необходимо учитывать при пересылке задач для выполнения на другие узлы.

Возрастание количества сообщений приводит к трем последствиям, влияющим на производительность системы:

– возрастание процессорного времени, расходуемого на маршрутизацию сообщений;

– возрастание процессорного времени, расходуемого на удовлетворение внешних запросов из локальных файлов данных;

– увеличение времени прохождения сообщений через промежуточные узлы.

Одно из основных требований диспетчеризации системы – ее эффективность. Эффективность использования ресурса определяется близостью загрузки к величине X^{max}. Перегрузка нежелательна, так как в этом случае резко увеличиваются затраты на пересылку информации и обслуживание распределенной системы, и производительность падает.

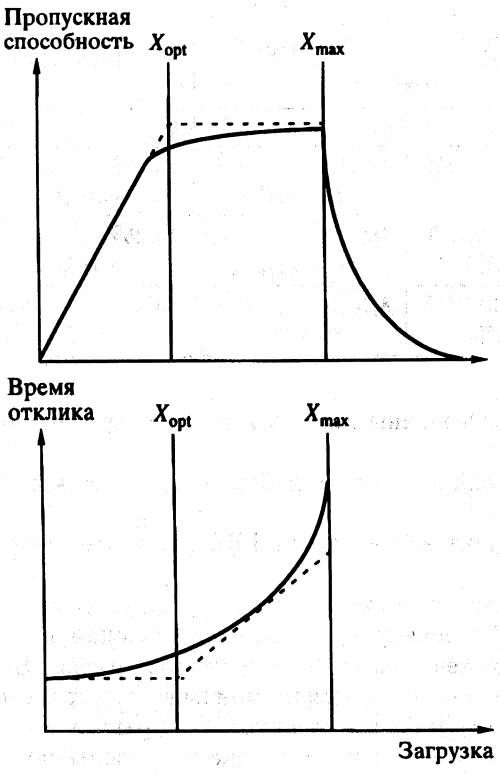


Рис. 13

На рис. 13 [54] показано изменение пропускной способности и времени отклика сети при увеличении нагрузки. Состояние X^{opt} соответствует такому значению загрузки, когда в сети начинают выстраиваться очереди. При этом пропускная способность растет незначительно, а время отклика начинает расти линейно. При дальнейшем росте загрузки и переполнении очередей (точка состояния X^{max}) пропускная способность резко падает, а время отклика – возрастает. Поэтому необходимо сформировать механизм, который стремился бы удержать состояние сети в окрестности точки X^{opt}.

Один из подходов заключается в выборе соответствующего алгоритма балансировки загрузки системы. В литературе этот алгоритм часто называют LBA (Load Bolancing Algorithm). Поясним этот подход на примере [32].

Пусть система состоит из четырех однородных компьютеров (рис. 8), на которых задачи распределяются, например так, как это было описано в разделе "Распределение агентов по процессорам". В системе существуют три алгоритма балансировки загрузки системы:

LBA-1. Не вмешивается в процесс балансировки.

LBA-2. Инициирует посылку сообщений. Узлы сети представляются связанными в кольцо и каждый узел может загрузить только соседний узел; загрузка прерывается, если один и тот же узел загружается дважды одной и той же задачей.

LBA-3. Инициирует прием сообщений, запрашивая загрузку (работу) у других узлов по очереди в случайном порядке.

Выбранный LBA функционирует по всей сети до тех пор, пока операционная система не передаст управление другому LBA. Считается, что LBA-2 требует меньше потерь на межпроцессорные связи, чем LBA-3 и цена загрузки зависит от расстояния между узлами.

Таблица 8

	LBA-1		LBA-2		LBA-3		Алгоритм переключения LBA	
	О	П	О	П	О	П	О	П
Лучший вариант	50%	22,5%	16,25%	20%	33,75%	78,75%	76,25%	60%
	36,25%		18,125%		56,25%		68,125%	
Средний вариант	8,75%	16,25%	75%	31,25%	16,25%	1,25%	13,75%	7,5%
	12,25%		53,125%		8,75%		10,65%	
Худший вариант	41,25%	61,25%	8,75%	48,75%	50%	20%	10%	32,5%
	51,25%		28,75%		35%		21,25%	

В качестве критериев выбраны только сокращение времени отклика и повышение пропускной способности.

Исходя из этих условий был создан набор правил, часть из которых приводится ниже:

1. Если общая загрузка низкая, тогда LBA-1 отвечает критерию минимизации времени отклика.

2. Если общая загрузка очень высокая и загрузка узлов системы очень неровная, тогда и LBA-2 и LBA-3 отвечают критерию минимизации времени отклика.

3. Если сильно загружен только один процессор, тогда LBA-3 особенно эффективно обеспечивает выполнение критерия минимизации времени отклика.

Остальные правила строятся аналогично. Эти правила легко могут быть трансформированы в правила вывода с лингвистическими переменными.

ЕСЛИ P1 – длина очереди маленькая
И P2 – длина очереди маленькая
И P3 – длина очереди маленькая
И P4 – длина очереди маленькая
ТОГДА LBA-1 эффект сокращения времени отклика положительный большой
И LBA-2 эффект сокращения времени отклика отрицательный большой
И LBA-3 эффект сокращения времени отклика отрицательный большой.

Согласно этому правилу вывода при отсутствии очередей на выполнение задач должны быть выбраны LBA-1. Аналогично строятся и другие правила вывода.

Описанный метод был промоделирован с помощью инструментальных средств [58, 59], результаты приведены в табл. 8.

В табл. 8 аббревиатура О означает время отклика, а аббревиатура П – пропускная способность. В табл. 8 показан процент лучших, худших и средних результатов по каждому из трех LBA, когда переключение режима одного LBA на другое не происходит. Последний столбец показывает процент лучших, худших и средних результатов, когда включен механизм выбора правил применения LBA и происходит выбор лучшего LBA для создавшейся ситуации. Между столбцами О и П показано их среднее значение.

Пример другого подхода, оптимизирующего загрузку системы поддержки принятия решений дан в [60, 61].

При изменении загрузки одного процессора ее величину будем определять управляющей функцией [60, 61]

$$X(t+h) = \begin{cases} a^+ + b^+ X(t) & \text{при увеличении нагрузки,} \\ a^- + b^- X(t) & \text{при уменьшении нагрузки,} \end{cases} \tag{10}$$

а для всей сети функцию (11) представим в виде

$$(11)\qquad \sum_{k=1}^{N} X_i(t+h) = \begin{cases} \sum\limits_{k=1}^{N} a_k^+ + \sum\limits_{k=1}^{N} b_k^+ X_k(t) & \text{при } \sum\limits_{k=1}^{N} X_k(t) < X^{\text{opt}}, \\ \sum\limits_{k=1}^{N} a_k^- + \sum\limits_{k=1}^{N} b_k^- X_k(t) & \text{при } \sum\limits_{k=1}^{N} X_i(t) > X^{\text{opt}}, \end{cases}$$

где a_k^+, b_k^+ – коэффициенты, используемые при увеличении нагрузки, a_k^-, b_k^- – при уменьшении, а индекс k означает номер узла.

Значения a_k^+, b_k^+, a_k^-, b_k^- определяются с учетом загрузки каждого узла, необходимых единиц загрузки для выполнения вновь поступивших и окончивших решение задач, приоритетов задач в узлах сети и других факторов, влияющих на ход вычислительного процесса.

Заметим, что диспетчер задач (или пользователь) одного узла не осведомлен о намерениях диспетчеров (или пользователей) других узлов.

Критериями для выбора той или иной стратегии действий могут служить принципы эффективности и равномерности использования ресурсов задачами эквивалентных классов (в литературе этот принцип называют принципом справедливости) [60].

Эффективность загрузки узлов сети определяется близостью общей загрузки на ресурсы к состоянию X^{opt} (см. рис. 13). Заметим, что эффективность зависит только от общей загрузки системы, поэтому различные размещения задач в узлах могут быть эффективными, пока общая загрузка близка к X^{opt}.

Если загрузка больше X^{opt}, т.е. $\sum\limits_{k=1}^{N} X_k(t) > X^{\text{opt}}$, то необходимо перейти в состояние, где $\sum\limits_{k=1}^{N} a_k^- + \sum\limits_{k=1}^{N} (b_k^- - 1) X_k(t) \leqslant 0$.

Значение параметров b_k^- должно удовлетворять соотношению

$$(12)\qquad \sum_{k=1}^{N} b_k^- = 1 - \frac{\sum\limits_{k=1}^{N} a_k^-}{\sum\limits_{k=1}^{N} X_k(t)}, \quad X_k(t) \leqslant X_k^{\text{opt}}.$$

Если $\sum\limits_{k=1}^{N} X_k(t)$ меньше X^{opt}, т.е. $\sum\limits_{k=1}^{N} X_k(t) < X^{\text{opt}}$, то значения параметров b_k^+ должны удовлетворять соотношениям:

$$\sum_{k=1}^{N} a_k^+ + \sum_{k=1}^{N} (b_k^+ + 1) X_k(t) \leqslant X^{\text{opt}},$$

$$\sum_{k=1}^{N} b_k^+ < \frac{X^{\text{opt}} - \sum\limits_{k=1}^{N} a_k^+}{\sum\limits_{k=1}^{N} X_k(t)} - 1.$$

Другим подходом к диспетчеризации является равномерное использование ресурсов задачами эквивалентных классов.

Этот подход заключается в том, что множество пользователей (или задач) разбивается на эквивалентные классы прав доступа к ресурсу. Пользователи (задачи), принадлежащие к эквивалентным классам, должны иметь равную долю ресурсов.

Критерий равномерности использования ресурсов может быть особенно полезен для задач, находящихся в фокусе внимания. Один из возможных критериев равномерности

$$F(X(t)) = \frac{\left(\sum\limits_{k=1}^{N} X_k(t)\right)^2}{N \sum\limits_{k=1}^{N} X_k^2(t)}.$$

Сходимость критерия равномерности определяется как стремление его значения к единице, т.е. $F(X(t)) \to 1$ при $t \to \infty$.

В терминах параметров увеличения (уменьшения) нагрузки это определяется следующими соотношениями:

$$\sum_{k=1}^{N} \frac{a_k^+}{b_k^+} \geqslant 0 \quad \text{и} \quad \sum_{k=1}^{N} \frac{a_k^-}{b_k^-} > 0 \tag{13}$$

или

$$\sum_{k=1}^{N} \frac{a_k^+}{b_k^+} > 0 \quad \text{и} \quad \sum_{k=1}^{N} \frac{a_k^-}{b_k^-} \geqslant 0. \tag{14}$$

В (13) равномерность загрузки растет при уменьшении загрузки и остается той же самой при ее увеличении. В (14) – наоборот, равномерность загрузки растет при увеличении загрузки и остается такой же при ее уменьшении.

Выражения (13) и (14) устанавливают, что как $\sum\limits_{k=1}^{N} a_k^+$ и $\sum\limits_{k=1}^{N} b_k^+$, так и $\sum\limits_{k=1}^{N} a_k^-$ и $\sum\limits_{k=1}^{N} b_k^-$ не должны иметь разные знаки.

Чтобы удовлетворить (13) и (14), значения $\sum\limits_{k=1}^{N} a_k^+$, $\sum\limits_{k=1}^{N} a_k^-$, $\sum\limits_{k=1}^{N} b_k^+$, $\sum\limits_{k=1}^{N} b_k^-$ должны быть положительны. Из (12) следует, что $\sum\limits_{k=1}^{N} b_k^-$ должно быть меньше 1. Следовательно, $\sum\limits_{k=1}^{N} a_k^+ \geqslant 0$, $\sum\limits_{k=1}^{N} a_k^- \geqslant 0$, $\sum\limits_{k=1}^{N} b_k^+ \geqslant 0$, $0 \leqslant \sum\limits_{k=1}^{N} b_k^- \leqslant 1$.

Таким образом, представление функции управления в виде (11) дает возможность оценить ограничения коэффициентов b_k^+, b_k^- для уже определенных a_k^+, a_k^-, при увеличении или уменьшении нагрузок в узлах сети, регулируя эффективность и равномерность загрузки при выполнении задач. Последнее имеет особенно большое значение для задач, находящихся в фокусе внимания.

В соответствии с данными о загрузке узлов (они могут быть сведены в общую таблицу загрузки системы, которая может функционировать как "доска объявлений"), распределенный диспетчер задач системы (фактически диспетчеры задач в узлах сети) может изменить уровень загрузки узлов, определяя и регулируя их через некоторый интервал времени h.

В начале каждого интервала узлы определяют свой уровень загрузки, создавая N-мерный вектор

$$X(t+h) = \{X_1(t+h), X_2(t+h), \ldots, X_n(t+h)\},$$

где n – число узлов (машин) в распределенной системе поддержки принятия решений.

Оценивая загрузку системы в целом и загрузку каждого ее узла, диспетчер для управления ходом вычислительного процесса в сети выполняет следующие функции [52]:

1. Находит задачи, которые не могут быть реализованы в своих узлах и определяет их приоритеты.

2. Находит узлы, не полностью загруженные и такие, у которых вероятность простоя процессоров достаточно велика.

3. Определяет число единиц загрузки, необходимых для выполнения каждой задачи, которая не может быть реализована в своем узле.

4. Определяет a_k^+, b_k^+, a_k^-, b_k^- для каждого узла.

5. Выделяет задачи, которые должны находиться в фокусе внимания и обеспечивает ресурсы для их реализации.

6. Перераспределяет, в случае необходимости, реализацию задач между узлами сети.

7. Обеспечивает выполнение требований эффективности и справедливости в ходе выполнения вычислительного процесса.

7. Заключение

1. Многоагентные системы являются объединением объектно-ориентированной технологии программирования и технологии искусственного интеллекта и являются очередным новым шагом в методах распределенного программирования.

2. Организация взаимодействия агентов в многоагентных системах, как всякая организация параллельных вычислений, является чрезвычайно сложной проблемой. Одна из определяющих характеристик распределенных вычислений – это совместное использование группы распределенных в пространстве, но связанных между собой, источников знаний (агентов), когда ни один из них не имеет необходимой информации и средств для решения всей задачи.

3. Современные аппаратные средства позволяют эффективно организовать взаимодействие в многоагентных системах.

4. Современные языки программирования и системы трансляции представляют разработчикам достаточно эффективные средства для создания программ, реализующих взаимодействия агентов в многоагентных системах.

5. В настоящее время предложен и реализован целый ряд методов, обеспечивающих управление вычислительным процессом и организацию взаимодействия агентов в многоагентных системах.

СПИСОК ЛИТЕРАТУРЫ

1. *Захаров В.Н.* Интеллектуальные системы управления: основные понятия и определения // Изв. РАН. Теория и системы управления. 1997. № 3. С. 138–145.
2. *Городецкий В.И.* Многоагентные системы: современное состояние исследований и перспективы // Новости искусственного интеллекта. 1996. № 1. С. 1–8.
3. *Serhrouchni A.E.F.* Multi-agent systems as a paradigm for intellegent system design // Informatica. 1997. V. 21. No. 2. P. 173–184.
4. *Bradshaw J.M., Duffield S., Benoit P., Woolley J.D.* KAoS: Toward an industrial – strength generic agent architecture. In Software Agents. Cambridge MA: AAAI/MIT Presss, 1996.
5. *Green S., Hurst L., Nangle B. et al.* Software agents: a review. 27 vay 1997 // http// www.cs.tcd.ie/research groups/ aig/iag/ pubreview.
6. *Трахтенгерц Э.А.* Методы генерации, оценки и согласования решений в распределенных системах поддержки принятия решений // АиТ. 1995. № 4. С. 3–32.

7. *Tockey S.* An example of agent-oriented systems // Northwest Artificial Intellegence Forum HomePage http: // www/halcyon.com./ topper/jv4n.1.htm.
8. *Maruyama K.* Concurrent object – oriented programming for distributed real-time systems // Information sciences. 1996. V. 93. No. 1, 2. P. 87–106.
9. *Wooldridge M., Jennings N.K.* Agent theories, architecture and languages: a survey // Lecture Notes in Artificial Intrlligence. V. 890. P. 1–39.
10. *Brazier F., Dunin-Keplitz B., Treur J., Verbrugge R.* Beliefs, Intentions and Desire // http: //ksi,cpsc.ucalgary.ca/KAW/KAW96/brazier/default.htmf
11. *Bradshaw J.M.* KAoS: An open agent architecture support reuse, interoperability and extensibility //http: // ksi.cpsc.ucalgary.ca/KAW/KAW96/ bradshaw/KAW/html.
12. *Palatnik M., Rosenschein J.S.* Long term constraints in multiagent negotiation // Distributed artificial intillegence. 13 Intern. Workshop. Technical Report. WS-94-02. AAAI-press. P. 244–259.
13. *Negraponte N.* The architecture machine: towards a more human envirenment. MIT Press, Cambridge. 1970.
14. *Maes P.* Agent that reduce work and information overload // Comm. of the ACM. 1994. V. 37. No. 7. P. 30–40.
15. *White J.* Telescript technology: the foundation of the electronic market place. General Magic white paper, 1995.
16. *Magedanz T.* Intelligent agent: state-of-the-art and potential application // 1 International Workshop on High Speed Networks and Open Distributed Platforms (Pre-proceedings) St. Petersburg, 1995.
17. *Трахтенгерц Э.А.* Протоколы локальных вычислительных сетей // АиТ. 1990. № 12. С. 3–40.
18. *Головкин Б.А.* Параллельные вычислительные системы. М.: Наука, 1980.
19. *Трахтенгерц Э.А.* Программирование параллельных процессов. М.: Наука, 1987.
20. *Богуславский Л.Б., Дрожжинов В.И.* Основы построения вычислительных сетей для автоматизированных систем. М.: Энергоатомиздат, 1990.
21. *Clemonty E., Logan D., Saarinem J.* ICAP/3090: parallel processing for large scale scientific and engineering problems // IBM System journal. 1988. V. 27. No. 4. P. 474–509.
22. *Crimsdal C.H.R.* Distributed operating systems for transputers. Microprocessors and microsystems. 1989. V. 13 No. 12. P. 79–88.
23. *Максимов К., Танаев А., Чубарков А.* Netscape Navigator – ваш путь в интернет. BHV – Санкт-Петербург, 1996.
24. *Ishida N.* Bridging humans via agent networks // Distributed artificial intillegence. 13 Intern. Workshop. Technical Report. WS-94-02. AAAI-press. P. 142–152.
25. *Dennis A.R., Abens N., Rom S., Nunamaker J.F.* Communication requirements and network evalution within electronic meeting system envirements //Decision support systems. 1991. V. 7. No. 1. P. 13–31.
26. Basic reference model for Open Systems Interconnection. ISO. 7498, 1983.
27. *Менн А.А.* Распределенные операционные системы управляющих вычислительных комплексов ЭВМ // АиТ. 1988. № 1. С. 8–37.
28. *Nunamaker J., Vogel D., Heiminger A., Martz B.* Experiences at IBM with group support systems: A field study // Decision support systems. 1989. No. 5. P. 183–196.
29. *Трахтенгерц Э.А.* Влияние архитектуры и структуры многопроцессорных вычислительных машин на языки программирования и методы трансляции // АиТ. 1986. № 3. С. 5–47.
30. *Барлетт Н., Лесли А., Симкин С.* Программирование на Java. Путеводитель. DiaSoft. Киев, 1996.
31. *Дэвис С.Р.* Программирование на Microsoft Visual Java++. Перевод с англ. Microsoft Press, 1997.
32. *Кен А., Гослинг Д.* Язык программирования Java. С.-Петербург. Питер Пресс. 1997.
33. *Cosling G., Mc Gilton H.* The Java language environment: A White Paper. Microsystems, 1995.

34. *Ousterhout J.* Tcl and the Tk Toolkit. Addison-Wesley, 1994.
35. *Cardelly L.* A language with distributed scope // Computing Systems. 1995. V. 8(1). P. 27–59.
36. *Kotz D., Cray R., Rus D.* Transportable agents support worldwide applications // Proccedings of the 7th ACM SIGOPS European Workshop, Sept. 1996. Connemara, Ireland.
37. *Dijkstra E.W.* Cooperating Sequential processes // Programing languages. 1968. No. 4. P. 43–112.
38. *Вегнер П.* Программирование на языке АДА. М.: Мир, 1983.
39. *Corkill D.D., Lesser V.L.* The use of meta-level control for coordination in a distributed problem solving network // Proc. of the 8-th IJCAI, 1983. P. 748–756.
40. *Dierkes S.* Load balancing with a fuzzy-decision algorithm // Information science 1997. V. 97. No. 1, 2. P. 159–178.
41. *Chen S.-J., Hwang C.-L.* Fuzzy multiple attribute decision making. Berlin: Springer-Verlag, 1992.
42. *Siskos J.L., Lochard J., Lombard J.* A multicriteria decision making metology under fuzziness: Application to the evalution of radiological protection in nuclear power plants // TIMS / studies in the Management Science. 1984. V. 20. P. 261–283.
43. *Zadeh L.A.* The concept of a lindustic variable and its approximate reasoning // Computer. 1988. V. 1. P. 83–93.
44. *Roy B.* Partial preference analysis and decision aid: The fuzzy outranking relation concept // In conflicting objectives and decisions. D.E. Bell, R.L. Kelney, R. Raiffa (eds). New York: Wiley, 1997. P. 40–75.
45. *Zimman H.J.* Fuzzy set, decision making and expert system. Boston: Kluwer, 1987.
46. *Brans J.P., Mareshal B., Vincke Ph.* Prometei. A new famille of outranking methods in multicriteria analysis // Operational Research'84. North-Holland, Amsterdam. (Proceeding of the tenth IFORS International conference on operational research, Washington D.C.). 1984. P. 477–490.
47. *Танака К.* Итоги рассмотрения факторов неопределенности и неясности в инженерном искусстве // Нечеткие множества и теория возможностей / Под ред. Р.Я. Ягера М.: Радио и связь, 1986. С. 37–50.
48. *Shen H.* Self-adjusting mapping: A heuristic mapping algorithm for mapping parallel programs into transputer networks // Proc. 1st OUG Developing Transputer Application. Edinburgh, Scotland, Sept. 1989.
49. *Snyder L., Berman F.* On mapping parallel algorithms into parallel architectures // J. Parallel and Distributed Computing. 1987. No. 4.
50. *Aggarwal J.K., Lee S.-Y.* A mapping strategy for parallel proccessing // IEEE Trans. Comput. 1987. C-36.
51. *Catenia V., Puliafito A., Vita L.* A Fuzzy approach to mapping problem // Information Sciences. 1996. V. 95. No. 3, 4. P. 191–218.
52. *Трахтенгерц Э.А.* Диспетчеризация задач в распределенных системах поддержки принятия решений // АиТ. 1996. № 8. С. 174–185.
53. *Гнеденко В.В., Коваленко И.М.* Введение в теорию массового обслуживания. М.: Наука, 1966.
54. *Борзенко В.И., Косых В.С., Трахтенгерц Э.А., Шершаков В.М.* Многокритериальное управление локальной вычислительной сетью с использованием переменного управления к среде передачи // АиТ. 1992. № 7. С. 154–164.
55. *Сантос Л.П., Гелмерс А., Проенса А.* Стратегия мониторинга интенсификации обмена сообщениями для параллельных систем с распределенной памятью // Программирование. 1995. № 1. С. 71–77.
56. *Aguilar J., Gelenbe E.* Task assignment and transaction clustering heuristics for distributed systems // Information Sciences. 1997. V. 97. No. 1, 2. P. 199–219.
57. *Park K.* Process migration in distributed operating systems // Trans. Inf. Proc. 1990. V. 31. P. 1080–1090.
58. University of Dortmund, Chair for Computer Science. V1, HI-Slang Reference manual, 1993.

59. University of Dortmund, Chair for Computer Science. V1, HII-GRAPHIC User's Guide, 1993.
60. *Jain R., Chin D.M., Hawe W.* A quantative measure of fairness and discrimination for resource allocation in shared systems. Technical report. Digital Equipment corporation, DEC-TR-301, 1988.
61. *Косых В.С.* Разработка и исследование методов оптимизации передачи информации в вычислительных сетях путем динамического регулирования загрузки. Диссертация на соискание ученой степени к.т.н. М.: ИПУ, 1994.

Поступила в редакцию 04.11.97